

This is a series of extracts from *Object Oriented Perl*, a new book from Manning Publications that will be available in August 1999. For more information on this book, see <http://www.manning.com/Conway/>.

What is object-oriented Perl?

Object-oriented Perl is a small amount of additional syntax and semantics, added to the existing imperative features of the Perl programming language. Those extras allow regular Perl packages, variables, and subroutines to behave like classes, objects, and methods.

It's also a small number of special variables, packages and modules, and a large number of new techniques, that together provide inheritance, data encapsulation, operator overloading, automated definition of commonly used methods, generic programming, multiply-dispatched polymorphism, and persistence.

It's an idiosyncratic, no-nonsense, demystified approach to object-oriented programming, with a typically Perlsh disregard for accepted rules and conventions. It draws inspiration (and sometimes syntax) from many different object-oriented predecessors, adapting their ideas to its own needs. It reuses and extends the functionality of existing Perl features, and in the process throws an entirely new slant on what they mean.

In other words, it's everything that regular Perl is, only object-oriented.

Using Perl makes object-oriented programming more enjoyable, and using object-oriented programming makes Perl more enjoyable too. Life is too short to endure the cultured bondage-and-discipline of Eiffel programming, or to wrestle the alligators that lurk in the muddy semantics of C++. Object-oriented Perl gives you all the power of those languages, with very few of their tribulations. And best of all, like regular Perl, it's fun!

Before we look at object orientation in Perl, let's talk about what object orientation is in general...

The essentials of object orientation

You really need to remember only five things to understand 90% of the theory of object orientation:

- an *object* is anything that provides a way to locate, access, modify, and secure data;
- a *class* is a description of what data is accessible through a particular kind of object, and how that data may be accessed;
- a *method* is the means by which an object's data is accessed, modified, or processed;
- *inheritance* is the way in which existing classes of object can be upgraded to provide additional data or methods;
- *polymorphism* is the way that distinct objects can respond differently to the same message, depending on the class they belong to;

This section discusses each of these ideas.

Objects

An object is an access mechanism for data. In most object-oriented languages that means that objects act as containers for data (or at least, containers for pointers to data). But in the more general sense, *anything* that provides access to data (a variable, a subroutine, a file handle) may be thought of as an object.

The various data to which an object provides access are known as its *attribute values*, and the containers storing those attribute values are called *attributes*. Attributes are (usually) nothing more than variables that have somehow been exclusively associated with a given object.

Objects are more than just collections of variables however. In most languages, objects have an extra property called *encapsulation*. Encapsulation¹ means that the attributes of an object are not directly accessible to the entire program. Instead, they can only be accessed through certain subroutines that are associated with the object. Those subroutines are called *methods*, and they are (usually) universally accessible. This layer of indirection means that methods can be used to limit the ways in which an object's attribute values may be accessed or changed. In other words, an object's attribute values can only be retrieved or modified in the ways permitted by that object's methods.

Let's take a real-world example of an object: an automated teller machine. An ATM is an object because it provides (controlled) access to certain attribute values, such as your account balance, or the bank's supply of cash. Some of those attribute values are stored in attributes within the machine itself (i.e. its cash trays), whilst others are stored elsewhere (i.e. in the bank's central accounts computer). From the client's point of view, it doesn't matter where the attribute values actually are, so long as they're accessible via the ATM object.

Access to the ATM's various attributes is restricted by the *interface* of the machine. That is, the various buttons, screens, and slots of the ATM control how encapsulated attribute values (cash, information, etc.) may be accessed. Those restrictions are designed to ensure that the object maintains a consistent internal state and that any external interactions with its attributes are valid and appropriate.

For example, most banks don't use ATMs consisting of a big basket of loose cash and a note pad on which you record exactly how much you took. Even if the bank could assume that everyone was honest, it couldn't assume that everyone was infallible. People would inevitably end up taking (or recording) the wrong amount by mistake, even if no-one did so deliberately.

The restrictions on access are in the client's interest too. The machine can provide access to attribute values that are private to a particular client (e.g. their account balance) and it shouldn't make that information available to just anyone. Even if we are pretending that all the ATMs clients are entirely honest, the account information shouldn't be universally

¹ *Encapsulation* is an awkward term, because it has two distinct meanings: "bundling things together" and "isolating things from the outside world". In the literature of object orientation both senses of the word have been used at different times. Originally, *encapsulation* was used in the "bundling" sense, as a synonym for *aggregation*. More recently, *encapsulation* has increasingly been used in the "isolation" sense, as a synonym for *data hiding*. It's in that more modern sense that the term is used hereafter.

available, because eventually someone will access and modify the wrong account data by accident.

In object-oriented programming, an object's methods provide the same kinds of protection for data. The question is: *how does an object know which methods to trust?*

Classes

Setting up an association between a particular kind of object and a set of trusted subroutines (i.e. methods) is the job of the object's *class*. A class is a formal specification of the attributes of a particular kind of object, and of the methods that may be called to access those attributes.

In other words, a class is a blueprint for a given kind of object. Every object belonging to a class has an identical interface (a common set of methods that may be called) and *implementation* (the actual code defining those methods and the attributes they access). Objects are said to be *instances* of the class.

When a program is asked to create an object of a particular kind, it consults the appropriate class definition (blueprint) to determine how to build such an object. Typically the class definition will specify what attributes the class's objects possess and where those attributes are stored (i.e. inside the object, or remotely through a pointer or reference).

When a particular method is called on an object, the program again consults the object's class definition to ensure that the method is "legal" for that object (i.e. the method is part of the object's blueprint), and that the method has been called correctly (i.e. in line with the definition in the class blueprint).

For example, in software controlling a bank's automated teller network there might be a class called `ATM` that describes the structure and behaviour of objects that represent individual ATMs. The `ATM` class might specify that each `ATM` object has the attributes `cash_remaining`, `transaction_list`, `cards_swallowed`, etc., and methods such as `start_up()`, `withdraw_cash()`, `list_transactions()`, `restrict_withdrawal()`, `chew_cards()`, `close_down()`.

Thereafter, when an `ATM` object receives a request to invoke a method called `withdraw_cash_without_debiting_account()`, it can check the `ATM` class blueprint and ascertain that the method cannot be called. Alternatively, if the (valid) method `close_down()` is defined to increment a (non-existent) attribute called `downtime`, then this coding error can be detected.

Class attributes and methods

So far, we've only considered attributes that are accessed through (i.e. "belong to") an individual object. Such attributes are more formally known as *object attributes*. Likewise, we've only talked about methods that were called on a particular object to manipulate its object attributes. No prizes for guessing that such methods are called *object methods*.

Unfortunately, object attributes and methods don't always provide an appropriate mechanism for controlling the data associated with the objects of a particular class. In particular, the attributes of an individual object of a class are not usually suitable for encapsulating data that belongs—collectively—to the complete set of objects of that class.

Let's go back to the `ATM` example for a moment. At the end of each day, the bank will want to know how much money in total has been dispensed from all its `ATM` machines.

Each of those machines will have a record of how much it has dispensed individually, but no machine will have a record of how much all the bank's machines have dispensed collectively. That information is not a property of a particular ATM. Rather, it's a collective property of the entire set of ATMs.

The most obvious solution is to design another kind of machine—an ATM coordinator—that gathers and stores the collective data of the set of ATMs (i.e. total cash dispensed, average number of transactions, funniest hidden video, etc.). We then create exactly one of these coordinator machines and arrange for each of the ATMs to feed data to it. Now we can access the accumulated ATM data through the interface of the coordinator machine.

In object-oriented terms, the design of the coordinator machine is the design of a separate class (say `ATM_Coordinator`), and the construction of such a machine corresponds to the creation of a single `ATM_Coordinator` object. This is certainly a viable solution to the problem of collective data, but it is unattractive in several respects.

For a start, this approach means that every time a class needs to handle collective data, we have to define yet another class and then create a single instance of it. Moreover, we have to be careful not to create *more* than one instance, to ensure that the collective data is not somehow duplicated or, worse still, fragmented.

Next, we have to provide some mechanism for connecting the collection of "individual" objects of the original class to the single object of the new "collective" class. That, in turn, means that the "collective" object has to be accessible anywhere that any "individual" object might be created. Hence the "collective" object must be globally accessible, which is generally considered a Bad Thing.

For these reasons, most object-oriented languages don't take this "helper class" approach to regulating collective data. Instead, they allow classes to specify a second kind of attribute, one that is "shared" by every object of that class, rather than being "owned" by a single object. Such attributes are, unimaginatively, called *class attributes*.

Of course, to maintain the appropriate protection for this kind of class-wide data², a class must also provide *class methods*, through which its class attributes may be safely accessed. A class method differs from an object method in that it is not called on a specific object (because, unlike an object attribute, a class attribute doesn't "belong" to a specific object). Instead, a class method is called on the class itself. This usually means that to call a class method we must specify both the class name and the method name (e.g. *invoke the `daily_total()` method for the class `ATM`*).

In some object-oriented languages, class methods provide *strong encapsulation* of a class's class attributes. In other words, there is no way to access a class attribute, except through the appropriate class method. Other languages offer only *weak encapsulation* of class attributes, by making them directly visible to any method of a class (i.e. to a class method or an object method). This means that class attributes may be accessed through individual objects as well. Perl enforces neither of these approaches, but allows us to use either or both.

² After all, the bank certainly doesn't want devices outside the ATM network accessing its collective ATM records.

Inheritance

If you're building an extension to your house, or customizing a car, or upgrading your computer, you normally start with an existing blueprint and add on (or replace) certain bits. If your original blueprints are good, it's a waste of time and resources to start from nothing and separately reconstruct nearly the same thing as you already have.

The same thing happens in object-oriented programming. Often you have a class of objects that partially meets your requirements (say a class that represents a truck), and you want to create a new class that exactly meets your needs (say a class that represents a fire-truck).

To produce a class representing fire-trucks, it's not necessary to code that class from scratch, reproducing (or maybe cutting and pasting) the original truck code, and then adding new methods to implement alarms, ladders, hoses, red braces, etc.

Instead, we can just tell the program that the new `FireTruck` class is based on (or *is derived from* or *inherits*) the existing `Truck` class. Then we tell it to add certain extra features (i.e. additional attributes and methods) to the `FireTruck` class, over and above those it inherited from the `Truck` class. Any class like `FireTruck` that inherits attributes and methods from another is called a *derived class* or sometimes a *child class*. The class from which it inherits (i.e. `Truck` in this case) is called its *base class* or its *parent class*.

The relationship between a base class and its derived class is called the *is-a* relationship, because an object of a derived class must necessarily have all the attributes and methods of an object of the base class, and hence it "is a" base-class object for all practical purposes. This idea corresponds to our inherent sense of the hierarchy of categories: a fire-truck *is-a* truck, an automated teller machine *is-a* machine, a hench-person *is-a* person, an unnecessarily long list of analogies *is-a* list of analogies.

The *is-a* relationship is *transitive*, so you can have increasingly general categories over more than two levels: a fire-truck *is-a* truck *is-a* vehicle *is-a* device *is-a* thing; a hench-person *is-a* person *is-a* animal *is-a* life-form *is-a* thing³.

Note however that the *is-a* relationship is not bi-directional. Though an object of a derived class is always an object of a base class, it's not always (or even usually) true that an object of a base class *is-a* object of a derived class. That is, although a fire-truck always *is-a* truck, it's not the case that a truck always *is-a* fire-truck.

Inheritance and abstraction

Naturally, having created a useful base class like `Truck`, we are immediately going to derive from it not just a `FireTruck` class, but also classes representing dump trucks, tow trucks, pickup trucks, armored cars, cement mixers, delivery vans, etc. Each of these will (separately) inherit the same set of characteristics from the original `Truck` class, but each will extend or modify those characteristics uniquely. The relationship between the `Truck` class and its numerous child classes is shown in Figure 1.

³ In fact, just about any class of object can be traced back to being a "thing". Of course, that doesn't mean we have to represent all those higher levels of categorization in an actual program. The universal "thing-ness" of a fire-truck, an ATM, or a hench-person is probably completely irrelevant to most applications.

Using inheritance means that we only have to specify how a fire-truck or a cement mixer or an armored car differs from a regular truck, rather than constantly needing to restate all the standard features of trucks as well. That makes the code that defines each type of truck shorter (assuming we already have the code for a truck).

More importantly, it reduces our maintenance load because any change to the behaviour of the general `Truck` class (for example, modifying its `register()` method in response to some change in transport regulations) is automatically propagated to all the specific truck classes (`FireTruck`, `DumpTruck`, `ArmoredCar`, etc.) that inherit from `Truck`.

In this way, inheritance also provides a way of capturing the *abstract relationships* between specific classes of object within a program. Thus, the class of fire-trucks is a special case of the more general class of trucks, which in turn might be a special case of the very general class of vehicles. The more abstract classes are generalized blueprints that define the common features of a wide range of kinds of objects. The more specialized classes presuppose those common features, and then describe the additional attributes and methods unique to their particular kind of object.

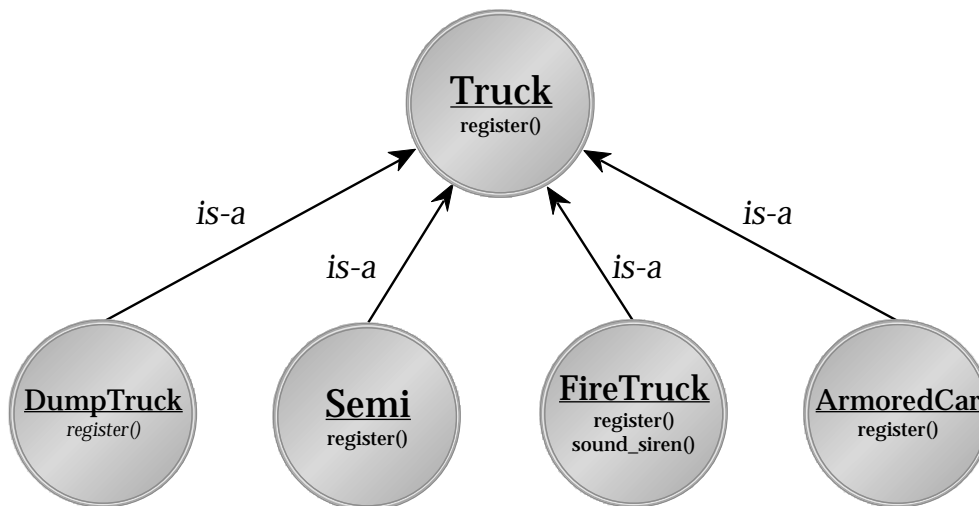


Figure 1: Inheriting from the Truck class

Inheritance hierarchies

The relative ease with which we can create and maintain new classes by inheriting from existing one will almost certainly encourage us to create more complex chains of inheritance. For example, there are many specialized types of fire-trucks: ladders, tankers, pumpers, snorkels, tarmac crash vehicles, etc. Likewise there are many species of dump truck: double bottom, highside end, lowside end, two-axle tractor, three-axle tractor, bob-tail, etc.

We may need individual classes for each of these very specific types of trucks, perhaps because each of them has unique regulations governing their registration and inspection. By

deriving such classes from `FireTruck` and `DumpTruck`, we might extend the set of class relationships shown in Figure 1 to the hierarchy shown in Figure 2.

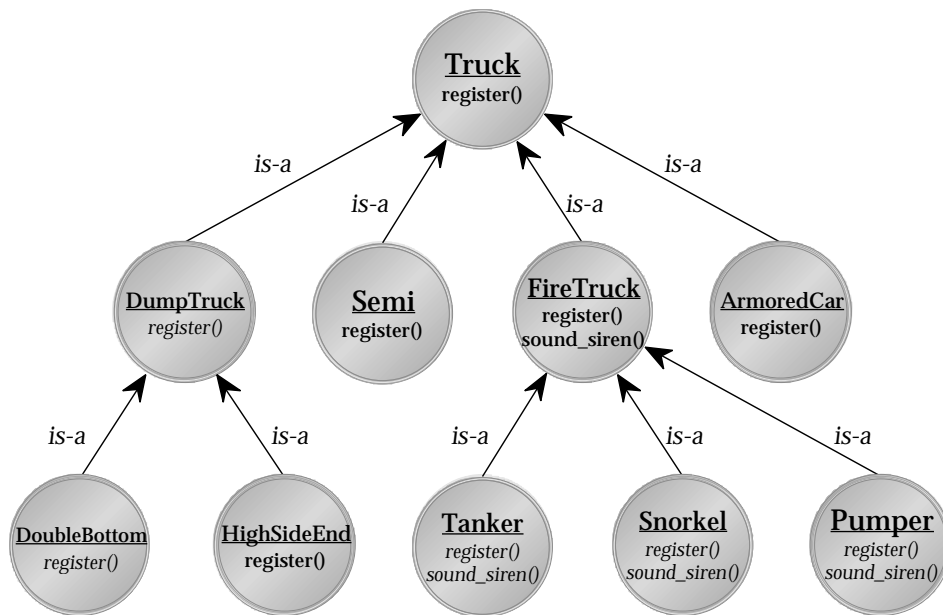


Figure 2: Extending the Truck hierarchy

Within such a hierarchy, every class offers all the methods offered by any class above it in the hierarchy. Therefore, objects of a particular class can always be treated as if they were objects of some class higher in the inheritance tree.

For example, both a `TankerFireTruck` object and a `DoubleBottomDumpTruck` object may be treated as if they were `Truck` objects (i.e. you could call their `register()` method), because both of them can trace their "ancestry" back to the primordial `Truck` class. However, of the two, only the `Tanker` object can be treated as a `FireTruck` object (i.e. you could call its `sound_siren()` method), because only the `Tanker` object can trace its ancestry to class `FireTruck`.

Note that some of the classes in the `Truck` hierarchy choose to redefine one or more of the methods they inherit (see the next section for an explanation of why they might want do that). For example, the `Semi` class redefines the `register()` method it inherits from class `Truck`. We can distinguish a method that has been (re-)defined in a class, from a method that a class merely inherits from its parent, by listing inherited methods in italics.

Polymorphism

If you've ever gone up to someone in a bar or club and introduced yourself, you know that people can respond to the very same message ("I'd like to get to know you better") in very different ways. If we were to categorize those ways, we could create several classes of `person`: `ReceptivePerson`, `IndifferentPerson`, `ShyPerson`, `RejectingPerson`, `RejectingWithExtremePrejudicePerson`, `JustPlainWeirdPerson`.

Turning that around, we can observe that the way in which a particular person will respond to your message depends on the kind of person they are. A `ReceptivePerson` will respond enthusiastically, a `ShyPerson` will respond tentatively, and a `JustPlainWeirdPerson` will probably respond in iambic pentameter. The original message is always the same; the response depends on the kind of person who receives it.

Language theorists⁴ call this type of behaviour *polymorphism*. When a method is called on a particular object, the actual method that's involved may depend on the class to which the object belongs. For instance, if we call an object's `ignite()` method, its response will be quite different depending on whether it belongs to the `Paper`, `Rocket`, `Passion`, or `FlameWar` class.

Randomly calling an identically named method on objects of different classes is not, of course, a recommended programming technique. However, polymorphic behaviour does prove extremely useful when there is some explicit relationship between the various classes of objects, or when there is an implicit relationship or a common universal property shared between them. The following subsections discuss each of these cases.

Inheritance polymorphism

Suppose we are creating an object-oriented system for tracking the registration and inspection of trucks. We'd almost certainly want to use our `Truck` class (and its many descendents) to implement the parts of the system that represent individual trucks.

Typically, the objects representing the various trucks would be collected in some kind of container, probably a list. Some operations will need to be carried out on individual objects (e.g. register this particular truck, schedule an inspection for that one, etc.), but many tasks will have to be performed on every truck in the system (e.g. send out an annual registration notice for each, print a complete list of recent inspection dates, etc.)

For operations that need to be performed on every truck, the application is likely to walk along the truck list using a loop, calling the appropriate method for each object in turn. For example, the loop might call each object's `print_registration_reminder()` method.

The problem is that the actual procedure to be followed by each object may be different, depending on the actual kind of truck the object represents (i.e. the actual class to which it belongs). For instance, the form for registering a semi-trailer may be very different from the one for a fire-truck, or for an armored car. If that's the case, the processing loop will have to determine the class of each object and then branch to perform a separate method call for each distinct class. That's a pain to code, and a bigger pain to re-code every time we add or remove another class of truck.

This situation is the ideal place to use polymorphism. If the ancestral `Truck` class has a `register()` method, then we are guaranteed that every derived class also has a `register()` method (i.e. the one that it inherits from `Truck`). However, when we specify the various derived classes, we may choose to *replace* the inherited `register()` method with one specific to the needs of the derived class.

Having given each class its own unique `register()` method, we can then walk the list of objects and simply call `register()` on each. We're sure each can respond to that method

⁴ ...most of whom live at ground-zero in the `JustPlainWeirdPerson` category...

call because at the very least they'll use the `register()` they inherited from the `Truck` class. However, if they have a more specialized way of registering themselves, then that more specialized method will be automatically invoked instead. In other words, we can arrange that each object has a `register()` method, but not necessarily the same `register()` method.

The result is that, although our loop code is very simple:

```
for each object in the list...
  call its register() method
```

the response to those calls is always appropriate to the particular object on which the method is called. Better yet, if we subsequently add a new class derived from `Truck`, and then put objects of that new class in the list, the old code will continue to work without modification. When the loop encounters an object of the new class, it will simply call that object's new `register()` method, and execute the new behaviour specified by the object's class definition. If the new class didn't define any new behaviour, the old behaviour inherited from class `Truck` will be used instead.

This kind of polymorphism is known as *inheritance polymorphism*, because the objects whose methods are called belong to a hierarchy of classes that are related by inheritance. The presence of the required method in the base class of the hierarchy ensures that objects of any derived class can always respond (if only generically) to a given method call. The ability to redefine individual methods in derived classes allows objects of those classes to respond more specifically to a particular method call, if they so wish.

All object-oriented languages support inheritance polymorphism⁵; for some, it's the only form of polymorphism they permit. But it certainly isn't the only form that's possible. In fact, there's no need for objects that are treated polymorphically to have any kind of class relationship at all.

Interface polymorphism

The alternative approach to polymorphism is to allow any object with a suitable method to respond to a call to that method. This is known as *interface polymorphism*, because the only requirement is that a particular object's interface provide a method of the appropriate name⁶.

For example, since there are probably no actual `Truck` objects used in the truck registry application, there's no real need for the `Truck` class at all (at least, as far as the polymorphism in the registration loop is concerned). So long as each object in the list belongs to a class that has a `register()` method, the loop doesn't really care what their ancestral class was (i.e. whether they are trucks, truckers, trucking companies, or truculents). Provided they can respond to a call on their `register()` method, the loop proceeds with serene indifference.

⁵ But that's rather a circular definition, since most language lawyers insist that this form of polymorphism is one of the essential characteristics a language must possess if it's to be considered object-oriented in the first place.

⁶ Statically-typed object-oriented languages (e.g. Java or Ada) usually also require that the argument list passed in the method call be type-compatible with the parameter list specified by the object's method.

Of course, that's a mighty big proviso. With inheritance polymorphism we could be sure that every object in the list did have a `register()` method (at the very least, the one it inherited from `Truck`). With interface polymorphism there's no such guarantee.

Worse still, because the list is almost certainly built at run-time, and modified as the program executes, unless we're very careful in setting up the logic of our application, we're not likely to know beforehand whether a particular object in the list can respond to a `register()` request. In fact, we're unlikely to find out until the application attempts to invoke the object's `register()` method, and finds that it doesn't have one.

Consequently, languages that allow interface polymorphism must also provide some run-time mechanism for handling cases where an object is unable to provide a requested method. Typically, this involves providing a means of specifying a "fall-back" subroutine that is called whenever an object cannot respond to a particular method invocation. Alternatively, such languages may have some form of exception system, and will trigger a well-defined exception (e.g. "No such method!") if the object cannot respond more appropriately.

Inheritance polymorphism is a special case of interface polymorphism, because a common base class guarantees that objects share a specific inherited method. Hence any language that supports interface polymorphism automatically supports inheritance polymorphism as well. As we shall see, Perl is such a language.

In order to understand object-oriented Perl, it's important to have a reasonable grasp of the language's non-object-oriented features. For example, the following extract explains the difference between package variables and lexicals—forever a source of confusion amongst novice Perl programmers...

Package variables

Perl variables come in two "flavours": *package variables* and *lexical variables*. They look and act much the same, but there are fundamental differences between them.

As the name suggests, each package variable belongs to a package (normally to the current one). Package variables are the ones that most casual Perl programmers use most of the time. They're the standard no-preparation-necessary, ready-to-serve, instant variables that are frequently used in small throw-away programs:

```
for ($i=0; $i<100; $i++)
{
    $time = localtime();
    print "$i at $time\n";
}
print "last time was: $time\n";
print "last index was: $ i\n";
```

Here, the variables `$time` and `$i` are both package variables. They are created automatically the first time they're referred to, and continue to exist until the end of the program. They belong to the current package (i.e. "main").

Whenever it's necessary to make a package variable's ownership explicit, its "personal" name can be *qualified* with the name of its package.

Package variables belonging to packages other than the current package are not accessible unless you use their fully-qualified name. For example, this code:

```
package main;

for ($i=0; $i<100; $i++)
{
    $Other_package::time = localtime();
    print "$i at $Other_package::time\n";
}

package Other_package;
print "last time was: $time\n";
print "last index was: $main::i\n";
```

uses the package variable `$time` belonging to the package called "Other_package", and the package variable `$i` belonging to the main package. Within their home packages, they can be referred to directly; elsewhere, you have to give their package name as well.

Note that the package name prefix always comes *after* the leading symbol. That is, you write `$Other_package::time`, not `Other_package::$time`.

Lexical variables

The other type of variable is a lexical variable. Unlike package variables, lexicals have to be explicitly declared, using the `my` keyword:

```
package main;

my $i;
for ($i=0; $i<100; $i++)
{
    my $time = localtime();
    print "$i at $time\n";
}
```

Lexical variables differ from package variables in that:

- They don't belong to any package (so you can't prefix them with a package name).
- They are only directly accessible within the physical boundaries of the code block or file in which they're declared. Hence in the code above, `$time` is only accessible to code that is physically located inside the `for` loop (and not to code that is called during or after that loop).
- They (usually) cease to exist each time the program leaves the code block in which they were declared. Hence in the code above, the variable `$time` ceases to exist at the end of each iteration of the `for` loop (and is recreated at the beginning of the next iteration).

It may help to think of the two types of variables (package and lexical) in the way the Ancient Greeks thought of their gods. They had big general-purpose gods like Uranus, Zeus, Aphrodite, and Atropos, who existed for all time and could appear anywhere without warning. These are analogous to package variables⁷.

⁷ The big Greek gods even came in "packages": `$Titans::Uranus`, `$Olympians::Zeus`, `$Olympians::Aphrodite`, `$Fates::Atropos`.

Then there were the small, specialized gods like the spirits of trees, or door-steps, or a hearth. They were restricted to a well-defined domain (a tree, a building, the fireplace) and existed only for a specific period (the life of the tree, the occupation of the building, the duration of a fire). These are like lexical variables: localized and transient.

Generally speaking, package variables are fine for very short programs, but cause problems in larger code. This is because they're accessible throughout the program source, which means that changes made at one point in the code can unexpectedly affect the program's behaviour elsewhere. The typical example is something like this:

```
package Recipe;

sub print_recipes
{
  for ($i=0; $i<@_; $i++)
  {
    print_ingredients($_[$i]);
    print_directions($_[$i]);
  }
}

sub print_ingredients
{
  for ($i=0; $i< $#recipes; $i++)
  {
    print $_[0]->{ingredients}[$i], "\n";
  }
}
```

The problem is that `$i` is a package variable (since it's not pre-declared as a lexical with a `my` declaration). That means that the subroutines `Recipe::print_recipes` and `Recipe::print_ingredients` both use the same package variable (`$Recipe::i`) in their respective for loops. So after `Recipe::print_ingredients` has been called from within `Recipe::print_recipe`, `$Recipe::i` will no longer contain the index of the current recipe. Instead, it will contain a number one greater than the number of ingredients of the current recipe (since that's the value left in it by the for loop in `Recipe::print_ingredients`).

If we'd used lexical variables instead:

```
package Recipe;

sub print_recipes
{
  for (my $i=0; $i<@_; $i++)
  {
    print_ingredients($_[$i]);
    print_directions($_[$i]);
  }
}

sub print_ingredients
{
  for (my $i=0; $i<@_; $i++)
  {
    print $_[0]->{ingredients}[$i], "\n";
  }
}
```

there would have been no unexpected interaction between the two subroutines⁸. Each lexical `$i` is distinct, unrelated to any other lexical `$i` (or to the package variable `$Recipe::i`, for that matter). Most importantly, each is confined to the body of the `for` loop in which it's declared.

The only problem is that, in Perl, lexical variables and package variables look the same, and since package variables can be conjured into existence just by mentioning them, this similarity can lead to subtle difficulties. For example, if we added an extra statement to the end of the loop timer shown earlier:

```
package main;

my $i;
for ($i=0; $i<100; $i++)
{
    my $time = localtime();
    print "$i at $time\n";
}
print "last time was: $time\n";
```

we'd find that the last line printed:

last time was:

That's because the *lexical* variable `$time` exists only inside the `for` loop, so Perl assumes that when we referred to `$time` outside the loop we meant the (undefined) package variable `$main::time`. This problem doesn't arise if you always put a `use strict` at the start of your code, because `use strict` requires that all package variables be fully qualified (to avoid just this kind of confusion).

Object-oriented programming in Perl can involve darker mysteries too, such as references, closures modules, and even the enigmatic `typeglob`...

Typeglobs

Typeglobs are amongst the most poorly understood features of Perl (right up there with closures, in fact). But, like closures, they're actually very easy to understand and use, once you unravel their mysterious syntax and their polymorphic behaviour.

Perl maintains separate namespaces for each package, and for each type of named construct within a package. Hence within a given package you can have the variables `$FILE`, `@FILE`, and `%FILE` as well as the subroutine `&FILE`. Best of all, you can use them all at the same time.

Unlike many other languages, where an identifier must be associated with exactly one thing in the symbol table, in Perl there's no confusion because each identifier has a unique prefix symbol indicating its type. In fact, they all live together in the very same entry of their package's symbol table, as Figure 3 illustrates.

Each Perl symbol table entry is like a "sampler" box of chocolates: you get a slot holding a reference to one scalar, a slot holding a reference to one array, a slot holding a reference to

⁸ An interaction of this kind between subroutines is known as *coupling*, and just as in real life, it can cause no end of difficulties.

one hash, and a slot holding a reference to one subroutine (as well slots holding references to one filehandle, and one format).

You can access an entire symbol table entry via a special piece of syntax called a *typeglob*⁹: `*symbol_name`. For example, to refer to the complete symbol table entry for anything that's called "FILE" (i.e. `$FILE`, `%FILE`, `&FILE`, etc.), we would use the typeglob `*FILE`. The slots in that symbol table entry would contain a reference to the package scalar variable `$FILE`, a reference to the package array variable `@FILE`, a reference to the package subroutine `&FILE`, etc.

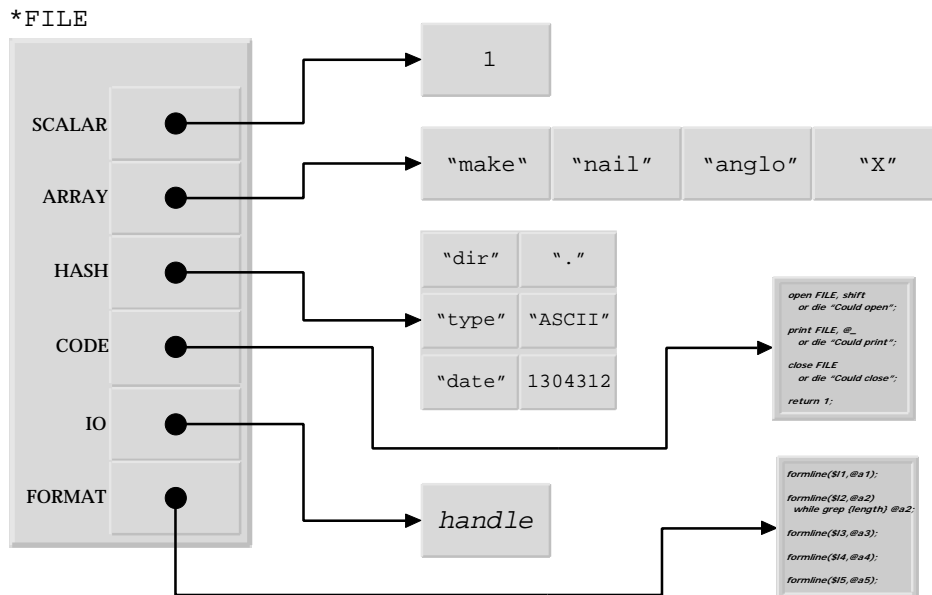


Figure 3: An entry (typeglob) in a package's symbol table

Once the necessary groundwork has been laid, we can finally start to talk about object-oriented programming in Perl...

Three little rules

If you've ever used another object-oriented programming language, or been traumatized by some prior exposure to object orientation, you're probably dreading tackling object orientation in Perl—more syntax, more semantics, more rules, more complexity. On the other hand, if you're entirely new to object orientation, you're likely to be equally nervous about all those unfamiliar concepts, and how you're going to keep them all straight in your

⁹ ...because it "globs" (generically matches) any type of variable with the correct name.

head while you learn the specific Perl syntax and semantics.

Relax!

Object-oriented Perl isn't like that at all. To do real, useful, production-strength, object-oriented programming in Perl you only need to learn about one extra function, one straightforward piece of additional syntax, and three very simple rules¹⁰. Let's start with the rules...

Rule 1: To create a class, build a package

Perl packages already have a number of class-like features:

- They collect related code together;
- They distinguish that code from unrelated code;
- They provide a separate namespace within the program, which keeps subroutine names from clashing with those in other packages;
- They have a name, which can be used to identify data and subroutines defined in the package.

In Perl, those features are sufficient to allow a package to act like a class.

Suppose we wanted to build an application to track faults in a system. Here's how to declare a class named "Bug" in Perl:

```
package Bug;
```

That's it! Of course, such a class isn't very interesting or useful, since it has no attributes or behaviour. And that brings us to the second rule...

Rule 2: To create a method, write a subroutine

Methods are just subroutines that are associated with a particular class. They exist specifically to operate on objects that are instances of that class.

Happily, in Perl a subroutine that is declared in a particular package is associated with that package. So to write a Perl method, we just write a subroutine within the package that is acting as our class.

For example, here's how we provide an object method to print our Bug objects:

```
package Bug;

sub print_me
{
    # The code needed to print the Bug goes here
}
```

Again, that's it. The subroutine `print_me` is now associated with the package `Bug`, so whenever we treat `Bug` as a class, Perl automatically treats `Bug::print_me` as a method.

Calling the `Bug::print_me` method involves that one extra piece of syntax—an extension to the existing Perl "arrow" notation. If you have a reference to an object of class

¹⁰ The three rules were originally formulated by Larry Wall, and appear in a slightly different form in the `perlobj` documentation.

Bug (we'll see how to get such a reference in a moment), you can access any method of that object by using a `->` symbol, followed by the name of the method.

For example, if the variable `$nextbug` holds a reference to a Bug object, you could call `Bug::print_me` on that object by writing:

```
package main;

# set $nextbug to refer to a Bug object, somehow, and then...

$nextbug->print_me();
```

Calling a method through an arrow should be very familiar to any C++ programmers; for the rest of us, it's at least consistent with other Perl usages:

```
$hsh_ref->{"key"};      # Access the hash referred to by $hashref
$arr_ref->[$index];    # Access the array referred to by $arrayref
$sub_ref->(@args);     # Access the sub referred to by $subref
$obj_ref->method(@args); # Access the object referred to by $objref
```

The only difference with the last case is that the thing referred to by `$objref` has many ways of being accessed (namely, its various methods). So, when we want to access an object, we have to specify which particular way (i.e. which method) should be used.

Just to be a little more flexible, Perl doesn't actually require that we "hard-code" the method name in the call. It's also possible to specify the method name as a scalar variable containing a string matching the name (i.e. a symbolic reference), or as a scalar variable containing a real reference to the subroutine in question. For example, instead of:

```
$nextbug->print_me();
```

we could write:

```
$method_name = "print_me";      # i.e. "symbolic reference" to some &print_me
$nextbug->{$method_name}();      # Method call via symbolic reference
```

or:

```
$method_ref = \&Bug::print_me;  # i.e. reference to &Bug::print_me
$nextbug->{$method_ref}();      # Method call via hard reference
```

In practice, the method name is almost always hard-coded.

When a method like `Bug::print_me` is called, the argument list that it receives begins with the object reference through which it was called¹¹, followed by any arguments that were explicitly given to the method. That means that calling `Bug::print_me("logfile")` is *not* the same as calling `$nextbug->print_me("logfile")`. In the first case, `print_me` is treated as a regular subroutine so the argument list passed to `Bug::print_me` is equivalent to:

```
( "logfile" )
```

In the second case, `print_me` is treated as a method so the argument list is equivalent to:

¹¹ The object on which the method is called is known as the *invoking object*, or sometimes the *message target*. It is the reference to this object that is passed as the first argument of any method invoked using the `->` notation.

```
( $objref, "logfile" )
```

Having a reference to the object passed as the first parameter is vital, because it means that the method then has access to the object on which it's supposed to operate¹². Hence you'll find that most methods in Perl start with something equivalent to this:

```
package Bug;

sub print_me
{
    my ($self) = shift;
    # The @_ array now stores the explicit argument list passed to &Bug::print_me
    # The rest of the &print_me method uses the data referred to by $self and
    # the explicit arguments (still in @_)
}
```

or, better still:

```
package Bug;

sub print_me
{
    my ($self, @args) = @_;
    # The @args array now stores the explicit argument list passed to &Bug::print_me
    # The rest of the &print_me method uses the data referred to by $self and
    # the explicit arguments (now in @args)
}
```

This second version is better because it provides a lexically scoped copy of the argument list (@args). Remember that the @_ array is "magical" in that changing any element of it actually changes the caller's version of the corresponding argument. Copying argument values to a lexical array like @args prevents nasty surprises of this kind, as well as improving the internal documentation of the subroutine (especially if a more meaningful name than @args is chosen).

The only remaining question is: *how do we create the invoking object in the first place?*

Rule 3: To create an object, bless a referent

Unlike other object-oriented languages, Perl doesn't require that an object be a special kind of record-like data structure. In fact, you can use *any* existing type of Perl variable—a scalar, an array, a hash, etc.—as an object in Perl¹³.

Hence, the issue isn't so much how to *create* the object (you create them exactly like any other Perl variable), but rather how to tell Perl that such an object *belongs* to a particular class. That brings us to that one extra built-in Perl function you need to know about. It's called `bless`, and its only job is to mark a variable as belonging to a particular class.

¹² There are similar automatic features in all object-oriented languages. C++ member functions have a pointer called `this`, Java member functions have a reference called `this`, Smalltalk methods have the `self` pseudo-object, and Python's methods (like Perl's) receive the invoking object as their first argument.

¹³ You can also bless other things, such as subroutines, regular expressions, and typeglobs.

The `bless` function takes two arguments: a reference to the variable to be marked, and a string containing the name of the class. It then sets an internal flag on the variable, indicating that it now belongs to the class¹⁴.

For example, suppose that `$nextbug` actually stores a reference to an anonymous hash:

```
$nextbug = {
    _id    => "00001",
    _type  => "fatal",
    _descr => "application does not compile",
};
```

To make turn that anonymous hash into an object of class `Bug` we write:

```
bless $nextbug, "Bug";
```

And, once again, that's it! The anonymous array referred to by `$nextbug` is now marked as being an object of class `Bug`. Note that the variable `$nextbug` itself hasn't been altered in any way (i.e. we didn't bless the *reference*); only the nameless hash it refers to has been marked (i.e. we blessed the *referent*). Figure 4 illustrates where the new class membership flag is set.

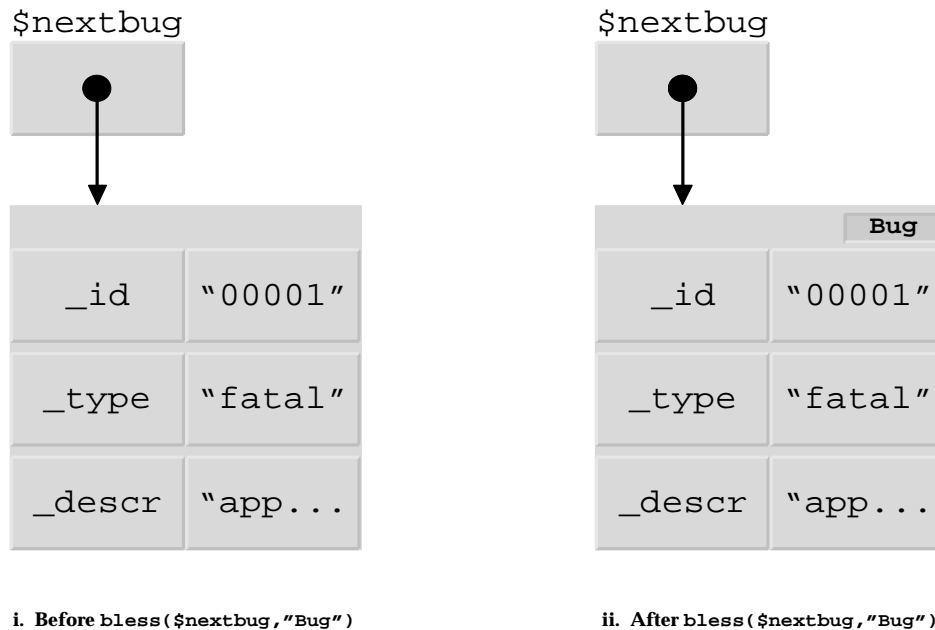


Figure 4: What changes when an object is blessed

¹⁴ Actually, the second argument is optional, and defaults to the name of the current package. However, although omitting the second argument may occasionally be convenient, it's *never* a good idea. Hence it's better to think of both arguments as being (morally) required, even if (legally) they're not.

You can check that the blessing succeeded by applying the built-in `ref` function to `$nextbug`. Normally when `ref` is applied to a reference, it returns the type of that reference. Hence, before `$nextbug` was blessed, `ref($nextbug)` would have returned the string `'HASH'`.

Once an object is blessed, `ref` returns the name of its class instead. So after the blessing, `ref($nextbug)` will return `'Bug'`. Of course the object itself still is a hash, but now it's a hash that *belongs* to the Bug class.

Perl classes use object methods to control access to encapsulated data. Typically such methods are named after the object attributes they provide access to. They usually take an optional argument through which new values may be assigned to a particular object attribute. But some attributes shouldn't be publicly writable and, unfortunately, Perl doesn't provide any built-in mechanism to enforce that. You have to resort to Psychology...

Catching attempts to change read-only attributes

Of course, because users of a class are often allowed to change some of an object's attributes by passing new values to the appropriate accessor methods, they may well expect to do the same with *all* the object's attribute values. This is not always the case and such a misgeneralization could lead to subtle logical errors in the program, since accessor methods for "read-only" attributes often simply ignore any extra parameters they are given.

There are several ways to address this potential source of errors. The most obvious solution is to resort to brute force, and simply kill any program that attempts to call a "read-only method" with arguments. For example:

```
package CD::Music;
use strict;
use Carp;

sub read_only
{
    croak "Can't change value of read-only attribute " . (caller 1)[3]
        if @_ > 1;
}

# read-only accessors
sub name      { &read_only; $_[0]->{_name} }
sub artist    { &read_only; $_[0]->{_artist} }
sub publisher { &read_only; $_[0]->{_publisher} }
sub ISBN      { &read_only; $_[0]->{_ISBN} }

# read-write accessors
sub last_played
{
    my ($self, $when) = @_;
    $self->{_played} = $when if @_ > 1;
    $self->{_played};
}

sub rating
{
    my ($self, $rating) = @_;
    $self->{_rating} = $rating if @_ > 1;
    $self->{_rating};
}
```

Here, each read-only access method calls the subroutine `CD::Music::read_only`, passing its original argument list (by using the "old-style" call syntax—a leading `&` and no parentheses). The `read_only` subroutine checks for extra arguments, and throws an informative exception if it finds any. Note that there will always be at least one argument to any method, namely the object reference through which the method was originally called.

Think of this technique as a form of Pavlovian conditioning for programmers: every time their code actually attempts to assign to a read-only attribute of your class, their program dies. Bad programmer!

As enjoyable as it may be to mess with people's minds in this way, this approach does have a drawback; it imposes an extra cost on each attempt to access a read-only attribute. Moreover, it isn't proactive in preventing users from making this type of mistake; it only trains them not repeat it, after the fact.

Besides, psychology has a much more subtle tool to offer us, in the form of a technique known as *affordances*¹⁵. Affordances are features of a user interface that make it physically or psychologically easier to do the right thing, than to do the wrong thing. For example, good architects don't put handles on unlatched doors that can only be pushed. Instead, they put a flat plate where the handle would otherwise be. Just about the only thing you can do with a plate is to push on it, so the physical structure of the plate helps you to operate the door correctly. In contrast, if you approach a door with a fixed handle, your natural tendency is to pull on it, which usually proves to be the right course of action.

Affordances work well in programming too. In this case, we want to make it psychologically awkward to attempt to change read-only object data. The best way to do that is to avoid raising the expectation that it is even possible in the first place.

For instance, we could change the names of the read-only methods to "get_..." and separate the two functions of each read-write accessor into distinct "get..." and "set..." methods:

```
package CD::Music;
use strict;

# read accessors
sub get_name      { $_[0]->{_name}      }
sub get_artist   { $_[0]->{_artist}   }
sub get_publisher { $_[0]->{_publisher} }
sub get_ISBN     { $_[0]->{_ISBN}     }
sub get_last_played{ $_[0]->{_publisher} }
sub get_rating   { $_[0]->{_rating}   }

# write accessors
sub set_last_played{ $_[0]->{_played} = $_[1] }
sub set_rating     { $_[0]->{_rating} = $_[1] }
```

Now the user of our class has no incentive to try to pass arguments to the read-only methods, because it doesn't make sense to do so. And because there are no `set_name`, `set_artist`, etc., it's obvious that these attributes can't be changed.

¹⁵ The concept of affordances comes from the work of user-interface guru Donald Norman. His landmark book *The Psychology of Everyday Things* (later renamed *The Design of Everyday Things*) is essential reading for anyone who creates interfaces of any kind, including interfaces to classes.

Most real object-oriented Perl classes use objects based on blessed hashes. But one of Perl's defining characteristics is flexibility, and in keeping with its unofficial motto—"There's more than one way to do it"—you can just as easily use any other Perl data type as the basis for object. There are at least six alternative ways of implementing a class: basing it on arrays, pseudo-hashes, scalars, anonymous subroutines, precompiled regular expressions, and typeglobs. But before considering some of the alternatives, it's important to explain why hashes aren't always the right choice...

What's wrong with a hash?

Hashes are well suited to act as the basis for objects. They can store multiple values of differing types, they give each value a descriptive label, they can be expanded to store additional items at need¹⁶, and they can be made hierarchical (by storing references to other anonymous hashes in an entry).

Hashes are usually a good choice for implementing class objects, but they're by no means perfect. For a start, they are a comparatively expensive way to store collections of data, occupying more space than an equivalent array, and providing slower access as well. Often those small overheads are insignificant, but occasionally (such as when large numbers of objects are involved, or when a much simpler data structure would do just as well) the difference in performance, or in style, matters.

A more serious problem with hashes has to do with an otherwise very convenient feature they possess, called *autovivification*. Autovivification is the name for what happens when you attempt to access a non-existent entry of a hash. Rather than complaining, Perl just automatically creates the missing hash entry for the key you specified, giving the new entry the value `undef`.

And that's the problem. If you have a reference to hash-based object (say, `$objref`), and you're using an attribute such as `$objref->{_weirdness_factor}`, then chances are that somewhere in the heat of coding, you'll accidentally write something like `$objref->{_wierdness_factor}++`.

The first time that code is executed, Perl won't complain about the spelling mistake or the fact that it causes your code to access a non-existent entry. Instead it will try to be helpful: autovivifying the new hash entry, then silently converting its `undef` value to zero, and finally incrementing it to 1. Thereafter you'll spend about a week trying to work out why that increment operator seems to increase the real world's weirdness factor, but not `$objref`'s.

One of those six alternate ways of implementing a class makes use of a feature only recently added to Perl—the pseudo-hash. The standard documentation on the feature is still rather terse, so the following explanation may be useful...

¹⁶ ...for example, if the class is later inherited by another...

Blessing a pseudo-hash

Neither a hash nor an array seems to provide the ideal basis for a Perl object. Hash entries are accessed by comprehensible keys, but hashes are big and slow. Arrays are compact and fast, but the use of integer indices can lead to obscure code. And both approaches are prone to autovivification-induced bugs. Ideally, we'd like the best of both worlds—fast access, compact storage, readable tags, and no autovivification.

A pseudo-what???

As of Perl release 5.005, that wish has been granted, in the form of a new (and experimental¹⁷) data structure called a *pseudo-hash*, which is really just an array reference that's pretending to be a hash reference.

To maintain the pretense, the array that's actually being referred to must have a reference to a real hash as its first element. That real hash is used to map key names onto array indices. In other words, a pseudo-hash has a structure like that shown in Figure 5, and is declared like this:

```
my $pseudo_hash = [ {a=>1,b=>2,c=>3}, "val a", "val b", "val c" ];
```

Such an array can still be accessed as an array, by specifying a numerical index in square brackets:

```
$pseudo_hash->[1];
```

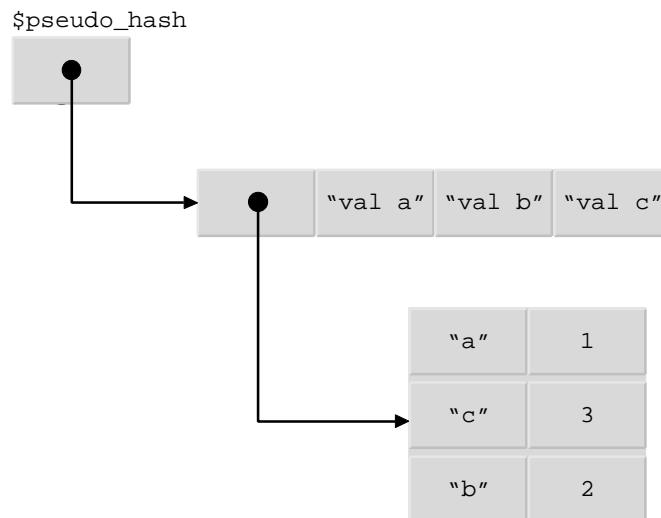


Figure 5: The structure of a pseudo-hash

¹⁷ Hence, if you're currently using a later version of Perl, you may need to check in the *perlref* documentation to see whether the details presented in this section are still correct.

But it can also be accessed as if it were a hash, by using one of the specified keys in curly braces:

```
$pseudo_hash->{"a"};
```

Whenever Perl encounters an array reference that is being used as a hash reference in this way, it translates the expression to something equivalent to the following:

```
$pseudo_hash->[$pseudo_hash->[0]->{"a"}];
```

In other words, it first retrieves the hash reference stored in element zero of the array (`$pseudo_hash->[0]`). It then uses that hash to look up the index corresponding to the specified key (`$pseudo_hash->[0]->{"a"}`), and finally it uses that index to access the appropriate element in the original array (`$pseudo_hash->[$pseudo_hash->[0]->{"a"}]`).

Limitations of a pseudo-hash

If the first element of a pseudo-hash array isn't a hash reference:

```
my $pseudo_hash = [ "not a hash ref", "val a", "val b", "val c" ];  
# and later...  
$pseudo_hash->{"a"};
```

then the program throws an exception with the message: **can't coerce array into hash**. If the first element is a hash reference, but the corresponding hash doesn't contain the given key:

```
my $pseudo_hash = [ {a=>1,b=>2,c=>3}, "val a", "val b", "val c" ];  
# and later...  
$pseudo_hash->{"z"};
```

then the program throws an exception with the message: **no such array field**¹⁸. In other words, unlike a real hash, pseudo-hash entries aren't autovivifying; they don't spring into existence the first time you attempt to access them.

You can add new entries to a pseudo-hash, but it's a two-step procedure. First you add a new key-to-index mapping:

```
$pseudo_hash->[0]->{"z"} = @{$pseudo_hash};
```

which maps the key "z" on to the first unused index in the pseudo-hash array. After that, you can access the new entry directly, to assign it a value:

```
$pseudo_hash->{"z"} = "value z";
```

Of course, if your stomach is strong enough, you can do those two steps in a single statement:

```
$pseudo_hash->[$pseudo_hash->[0]{"z"} = @{$pseudo_hash}] = "value z";
```

¹⁸ The reason it refers to a "field" instead of an "entry" will become clear in a moment.

Advantages of a pseudo-hash

The awkwardness of having to "manually" add new keys to a pseudo-hash is actually a useful property, because it helps to prevent hard-to-detect bugs that can easily find their way into classes built on ordinary hashes. Consider the `Transceiver` class defined in Figure 6. The class provides sentinel methods (`start_transmit` and `end_transmit`, `start_receive` and `end_receive`) that may be used to ensure that transmission and reception do not overlap.

```
package Transceiver;
$VERSION = 1.00;
use strict;

sub new
{
    my $class = ref($_[0])||$_[0];
    my $self = { receive=>0, transmit=>0 };
    bless $self, $class;
}

sub start_transmit
{
    my ($self) = @_;
    ++$self->{transmit} unless $self->{recieve};
    return $self->{transmit};
}

sub end_transmit
{
    my ($self) = @_;
    --$self->{transmit};
}

sub start_receive
{
    my ($self) = @_;
    ++$self->{receive} unless $self->{transmit};
    return $self->{receive};
}

sub end_receive
{
    my ($self) = @_;
    --$self->{receive};
}
```

Figure 6: A simple hash-based transceiver class

The problem is that the `Transceiver::transmit` method has accidentally been coded to check the status of the hash entry `$self->{recieve}` (instead of `$self->{receive}`). The first time it does so, this non-existent entry will produce a value of `undef`. Hence the `unless` test will *never* fail and transmission will always be allowed, no matter what the current state of reception is.

If we had implemented `Transceiver` objects as pseudo-hashes instead:

```

package Transceiver;
use strict;

sub Transceiver::new
{
    my $class = ref($_[0])||$_[0];
    my $self = [ {receive=>1, transmit=>2} ];
    $self->{transmit} = 0;
    $self->{receive} = 0;
    bless $self, $class;
}

# etc. as before

```

then the first time `Transceiver::transmit` was called, we would get an exception indicating: **No such array field...**, which would eventually lead us to the misspelled key.

Another unusual, but interesting choice of datatype for building classes is the scalar variable. Scalars can only hold a single value, which would seem like a serious limitation for an object. But sometimes less is more....

Blessing a scalar

You almost never see a Perl class that is based on a blessed scalar value. Although there are several good reasons for that, a scalar can occasionally prove to be the best choice for implementing an object.

The main reason that a scalar so rarely forms the basis of a Perl class, is that classes so rarely store only a single piece of information. One of the main reasons for building a class is to bind together a set of related attribute values and then provide controlled access to them. If the data is really only a single datum, then building an object-oriented shell around it usually seems like serious overkill.

In Perl we can't even use the excuse that data ought to be encapsulated, since Perl's encapsulation is almost entirely voluntary. If we have blessed a scalar and are passing around a reference to it (as `$sref`), there's absolutely nothing to prevent any part of the program completely ignoring the lovely controlled object-oriented interface we provided, and just manipulating the underlying scalar directly:

```

$$sref = undef; # Bwah-ha-ha-ha!!!

```

What's more, in those few cases where an object *does* only possess a single value, it's just as easy to go with a more familiar hash-based implementation, using only a single entry. Allocating an entire hash to store a single value may be considerably less efficient (both in terms of memory usage and access speed), but it has the advantages of:

- *Familiarity to the implementer.* The selection of a hash as the underlying object representation is often the automatic choice, and frequently not even a conscious one.
- *Familiarity to others.* A better reason for choosing a hash when a scalar would suffice, is that the hash-based implementation is also likely to be far more familiar to anyone else attempting to understand or modify the code.

- *Readability*. If nothing else, storing the single value as a hash entry means that the value has to be given a meaningful key, which ought to improve the code's readability.
- *Extensibility in subclasses*. After all, a class can never be sure that an internal representation sufficient to its own needs will serve derived classes equally well.

An object-oriented password

Despite all those factors against the practice, there's nothing immoral or illegal about blessing a scalar. In most cases, it's even slimming¹⁹. For example, Figure 7 illustrates the simple case of a class that implements an encrypted password as a single scalar string.

```
package Password;
$VERSION = 1.00;
use strict;

my @salt = ("A".."Z","a".."z","0".."9","/",".");

sub new
{
    my ($class, $cleartext) = @_;
    my $$salt = $salt[rand @salt].$salt[rand @salt];
    my $pw = crypt($cleartext,$salt);
    return bless \$pw, ref($class)||$class;
}

sub verify
{
    my ($self, $candidate) = @_;
    my $$salt = substr($$self,0,2);
    return crypt($candidate,$salt) eq $$self;
}
```

Figure 7: A scalar-based password class

The only tricky part about using scalars as objects is how to create one in the first place. Unlike arrays and hashes, scalars are not provided with a special syntax for creating anonymous instances. There's no syntax corresponding to [...] (which creates anonymous arrays), or to {...} (which creates anonymous hashes). Instead, we have to hijack a lexical variable (e.g. \$pw in the Password constructor).

The constructor takes a text string as its argument, randomly creates a "salt" value²⁰, encrypts the string with a call to the in-built crypt function, assigns the encrypted version to a lexical variable \$pw, and then blesses \$pw into the class.

¹⁹ "Reads faster, less memory!"

²⁰ The crypt function implements a family of related one-way encryption schemes. The actual scheme crypt uses is determined by a two-character "salt" string that is passed as its second argument.

The important point to understand is that, even though `$pw` is a lexical, it does not cease to exist at the end of the call to `Password::new`. That's because `bless` returns a reference to `$pw` and that reference is then returned as the result of `new`.

Assuming that the reference is immediately assigned to a variable in some outer scope:

```
my $password = Password->new("fermat");
```

then the number of "live" references to the scalar remains greater than zero, and the lexical scalar escapes destruction at the end of the scope in which it was declared.

The `verify` method is equally straightforward. It encrypts the candidate string and compares the result to the password string (i.e. to the invoking object itself). This process takes advantage of the fact that the first two letters of a `crypt`'ed string are identical to the "salt" with which the original call to `crypt` was seasoned.

Note that accessing the object's data is slightly different when the object is a scalar. We can't use the arrow notation to access an entry or an element (as we do with references to hashes or arrays). With a scalar-based object, we need to explicitly dereference the scalar reference. Thus the single value stored in the object referred to by `$self`, is always accessed as `$$self`.

The class could be used like so:

```
use Password;

print "Enter password: ";
my $password = Password->new(scalar <>);

# and later...

while (<>)
{
    last if $password->verify($_);
    print "Sorry. Try again: ";
}
```

It could reasonably be argued that the use of object orientation in this implementation is needlessly ostentatious. However, good software engineering practice would suggest that the mechanics of password creation and verification should be encapsulated in subroutines. Suppose, for example, that we later decide that the `crypt` algorithm is insufficiently secure, and that MD5 or PGP or SHA must be used instead? Clearly, we don't want raw calls to `crypt` spread throughout the code, to be hunted down and changed one at a time.

Class inheritance can be very complicated in some object-oriented languages, but Perl strips the concept back to a surprisingly simple idea: that inheritance tells an object where to look next..

How Perl handles inheritance

Perl's approach to inheritance is typically low-key and uncomplicated. Packages that are acting as classes simply announce their "allegiance" to some other class, and dynamically inherit all its methods. Perl also provides some standard methods that all classes inherit, and a small dose of syntactic sugar to make rewriting inherited methods easier. Let's start with the pledge of allegiance...

The @ISA array

A class informs Perl that it wishes to inherit from another class by adding the name of that other class to its @ISA package variable. For example, the class PerlGuru could specify that it wishes to inherit from class PerlHacker as follows:

```
package PerlGuru;
@ISA = ( "PerlHacker" );
```

And that's it. From that point on, whenever Perl needs to determine if PerlGuru has any inherited methods, it checks the contents of the array @PerlGuru::ISA. Any package whose name appears in that array is considered to be a parent class of PerlGuru. Of course, since it's an array, we can have many class names in @PerlGuru::ISA, allowing the class to inherit methods from more than one parent:

```
package PerlGuru;
@ISA = qw( PerlHacker LanguageMaestro Educator PunMeister );
```

And, of course, if those four parent classes also inherited from other classes:

```
package PerlHacker;
@ISA = qw( Programmer Obfuscator );

package PunMeister;
@ISA = qw( Writer Humorist OneSickPuppy );
```

then PerlGuru would also inherit methods from those "grandparents". All this inheritance creates the hierarchy shown in Figure 8.

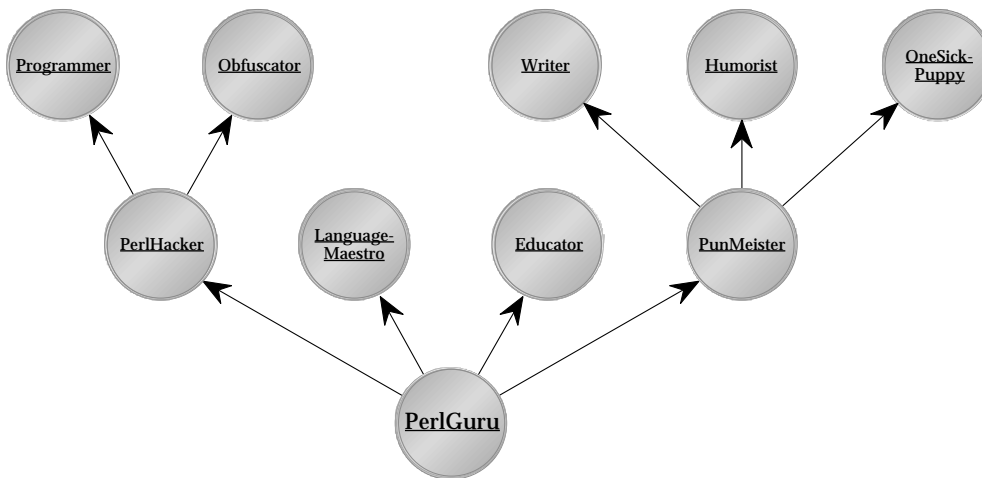


Figure 8: PerlGuru's inheritance hierarchy

What inheritance means in Perl

Inheritance in Perl is a much more casual affair than in other object-oriented languages. In essence, inheritance means nothing more than: *if you can't find the method requested in an object's blessed class, look for it in the classes that blessed class inherits from.*

In other words, if we call:

```
my $guru = PerlGuru->new();  
  
# and later...  
  
my $question = <>;  
print $guru->answer($question);
```

then, if class `PerlGuru` doesn't provide a `PerlGuru::answer` method, Perl starts trying the various parent classes (as specified by the current value of the `@PerlGuru::ISA` array). The parents are searched in a depth-first recursive sequence²¹, so Perl would look for one of the following (in this order):

- `&PerlGuru::answer` (look in the actual class of `$guru`),
 - `&PerlHacker::answer` (look in the class specified by the first entry in the variable `@PerlGuru::ISA`),
 - `&Programmer::answer` (look in the class specified by the first entry in the variable `@PerlHacker::ISA`)
 - `&Obfuscator::answer` (look in the class specified by the second entry in the variable `@PerlHacker::ISA`),
 - `&LanguageMaestro::answer` (look in the class specified by the second entry in the variable `@PerlGuru::ISA`),
 - `&Educator::answer` (look in the class specified by the third entry in the variable `@PerlGuru::ISA`),
 - `&PunMeister::answer` (look in the class specified by the fourth entry in the variable `@PerlGuru::ISA`),
 - `&Writer::answer` (look in the class specified by the first entry in the variable `@PunMeister::ISA`),
 - `&Humorist::answer` (look in the class specified by the second entry in the variable in `@PunMeister::ISA`),
 - `&OneSickPuppy::answer` (look in the class specified by the third entry in the variable `@PunMeister::ISA`).

If any of these methods is defined, the search terminates at once and that method is immediately called²². This process of searching for the right method to call is known as *method dispatch*.

²¹ Sean M. Burke's `Class::ISA` module (available from the CPAN) allows you to extract the exact sequence in which a class's parent's are searched, as a list of class names.

²² Note that, when looking in a parent class, Perl checks the left-most parent first, and then checks the left-most parent of that class, and the left-most parent of *that* class etc. Hence, if a class's left-most great-great-great-grandparent has a method of the right name (e.g. `answer`) then that method will be called, even if another of the object's direct parents also has a suitable method. In other words, you don't necessarily get the method that is "closest" up the inheritance hierarchy; you get the method that was inherited through the left-most inheritance chain. This is known as "left-most ancestor wins".

If you're used to the complicated inheritance semantics in some other object-oriented language, it's important to realize that inheritance in Perl is merely a way of specifying where else to look for a method, *and nothing else!* There is no direct inheritance of attributes (unless you arrange for it), nor any hierarchical calling of constructors or destructors (unless you explicitly write those methods that way), nor any compile-time consistency checks of the interface or implementation of derived classes.

This process of finding the correct method to call also explains why Perl ignores any prototype associated with a method, and why you can't use prototypes to constrain the number of arguments given to a method. The prototype check occurs when the code is being compiled, but at that point the compiler has no idea which of the many potential `answer` subroutines will actually be called (since it will depend on the contents of the various `@ISA` arrays at the time the method is actually called). So the compiler has no way of determining which subroutine's prototype to check the argument list against.

Where the call goes

The exact semantics of where (and in what order) Perl looks for a method are relatively straightforward, but warrant a brief discussion. The rules for handling a call such as `$obj->method()` can be summarized as follows:

1. If the class into which `$obj`'s referent is blessed (say `MyClass`) has a subroutine `method`, call that.
2. Otherwise, if `MyClass` has an `@ISA` array, step through each parent class in that array and apply steps 1 and 2 to it (i.e. recursively search in depth-first, left-to-right order up the hierarchy). If a suitable `method` subroutine is found in any package in the hierarchy, call that.
3. Otherwise, if the `UNIVERSAL` class has a subroutine `method`, call that.
4. Otherwise, if `MyClass` has an `AUTOLOAD` method, call that.
5. Otherwise, if one of the ancestral classes of `$obj`'s referent (once again searched in depth-first, left-to-right order) has an `AUTOLOAD` method, call that.
6. Otherwise, if the `UNIVERSAL` class has an `AUTOLOAD` method, call that.
7. Otherwise, give up and throw an exception: **Can't locate object method "method" via package "MyClass"**.

Once a suitable method has been found for an object of a particular class, a reference to it is cached within the class. Thereafter, any subsequent call to the same method through objects of the same class doesn't need to repeat the search. Instead, it uses the cached reference to go directly to the appropriate method.

If the class's `@ISA` array (or that of any of its ancestors) is modified, or if new methods are defined somewhere in the hierarchy, then the cached method may no longer be correct. In such cases, the cache is automatically cleared and the next method call simply does a new search (and, of course, re-caches the resulting subroutine reference).

The biggest hurdle that most budding object-oriented programmers face is coming to grips with polymorphism. But Perl removes the mystery (and the misery) with its straightforward and pragmatic approach...

Polymorphism in Perl

Those of us who hate having injections usually appreciate when our doctor says: "Okay, I'll count to 3: ...1...2...<jab!>...3", and the nastiness is over before it begins.

Guess what.

If you've been apprehensive about this section—either because you've heard polymorphism is "difficult", or because you've had trouble with it in other languages—you can relax. The nastiness is over. If you've read this far, you've already seen everything you need to know about polymorphism. Whilst some object-oriented languages have special syntaxes and a long list of rules, constraints, and conditions on the use of polymorphic methods, as you'll have realized by now, Perl has a different attitude.

In Perl, every method of every class is (potentially) polymorphic, as a direct consequence of the way that methods are automatically dispatched up the class hierarchy. There's no special syntax, no requirement for type-compatibility of method arguments, no need for inheritance relationships between classes. Just define your method, redefine it in any derived classes that need to act differently, and without even knowing it you're polymorphizing.

Interface polymorphism

Suppose we have an object reference (say, `$datum`) and we call a method (say, `print_me`) on it:

```
foreach my $datum ( @data )
{
    $datum->print_me();
}
```

The method dispatch mechanism determines the class of the invoking object (i.e. of `$datum`), and then looks in the corresponding package for a method of the appropriate name (i.e. `print_me`). Provided the object belongs to a class with a method named `print_me`, the method call succeeds and some action is taken. That action depends on the class of the invoking object, even though the call syntax is always the same.

The elements in the `@data` array might have been blessed into completely unrelated classes:

```
my @data = (
    GIF_Image->new(file=>"camelopard.gif", format=>"interlaced"),
    XML::File->new("./lamasery.xml"),
    PGP_Coded->new("Software is *not* a munition!"),
    HTTP::Get->new("http://www.perl.org/news.html"),
    Signature->new(),
);
```

but the same method call (i.e. `$datum->print_me()`) handles them all appropriately, so long as each object's class's interface provides a `print_me` method. That's known as *interface polymorphism*.

Inheritance polymorphism

Of course, the dispatch mechanism also has a fall-back strategy if the class of the invoking object *doesn't* provide a matching method. As explained earlier, it immediately searches through the object's ancestor classes, trying to find an inherited method with the correct name.

This means that if the object belongs to a class that *inherits* a method named `print_me`, the method call succeeds and some action is taken. Once again, that action depends on the class of the invoking object (or more accurately, on the "genealogy" of that class), even though the call syntax is still always the same. That's known as *inheritance polymorphism*.

Like some other object-oriented languages, Perl has a mechanism that allows class designers to redefine the behaviour of its standard arithmetic, logical, and other operators. Some software engineering purists decry such facilities, but there are good reasons for wanting them...

The problem

One aspect of object-oriented programming that seems to turn some people away is the need to call methods on objects, rather than manipulating the objects directly. It's not so much the efficiency of so many function calls (although that can be a concern too); it's the sheer ugliness of the code they produce.

Take Mark Biggar's standard `Math::BigFloat` module for example²³. `Math::BigFloat` objects store floating point numbers as character strings, and provide a range of methods for manipulating those string representations: `fneg` to negate them, `fadd` to add them, `fmul` to multiply them, etc.

We could use those methods to work out some calculation involving large numbers, such as the expected difference in per-capita gross domestic product between China and the USA in 1998²⁴. Given the most recent available statistics (i.e. for 1997):

```
%China =
(
  pop      => Math::BigFloat->new("1 221 591 778"),      # people
  gdp      => Math::BigFloat->new("3 390 000 000 000"),   # US dollars
  pop_incr => Math::BigFloat->new("1.0093"),             # annual % change
  gdp_incr => Math::BigFloat->new("1.097"),             # annual % change
);

%USA =
(
  pop      => Math::BigFloat->new("267 954 764"),        # people
  gdp      => Math::BigFloat->new("7 610 000 000 000"),   # US dollars
  pop_incr => Math::BigFloat->new("1.0087"),             # annual % change
);
```

²³ Not that there's anything inherently wrong with the `Math::BigFloat` package! On the contrary, it's well-implemented and very useful. We're just going to use it inappropriately in order to make a point about method-based operations in general.

²⁴ US\$25,814.89, in case you actually needed to know.

```

    gdp_incr    => Math::BigFloat->new("1.024"),          # annual % change
);

```

the following calculation is required:

```

$diff = Math::BigFloat->new((Math::BigFloat->new((Math::BigFloat->
    new((Math::BigFloat->new($China{gdp}->fmul($China{gdp_incr}))
    )->fdiv(Math::BigFloat->new($China{pop}->fmul($China{pop_incr}
    )))))->fsub(Math::BigFloat->new((Math::BigFloat->new($USA{gdp}
    ->fmul($USA{gdp_incr})))>fdiv(Math::BigFloat->new($USA{pop}->
    fmul($USA{pop_incr})))))))->fabs());

```

Yuck. Even breaking up the computation doesn't help the readability much:

```

$cpop    = Math::BigFloat->new( $China{pop}->fmul($China{pop_incr}) );
$cgdp    = Math::BigFloat->new( $China{gdp}->fmul($China{gdp_incr}) );
$upop    = Math::BigFloat->new( $USA{pop}->fmul($USA{pop_incr}) );
$ugdp    = Math::BigFloat->new( $USA{gdp}->fmul($USA{gdp_incr}) );
$cgdp_pc = Math::BigFloat->new( $cgdp->fdiv($cpop) );
$ugdp_pc = Math::BigFloat->new( $ugdp->fdiv($upop) );
$sdiff   = Math::BigFloat->new( $cgdp_pc->fsub($ugdp_pc) );
$diff    = Math::BigFloat->new( $sdiff->fabs() );

```

The standard method-based object-oriented interface just doesn't work here, because the numerous method calls swamp the meaning of the code in a sea of arrows, parentheses, and constructors. What we'd really *like* to be able to write is something like:

```

$diff =
    abs(($China{gdp} * $China{gdp_incr}) / ($China{pop} * $China{pop_incr})
    - ($USA{gdp} * $USA{gdp_incr}) / ($USA{pop} * $USA{pop_incr}));

```

which is at least *decipherable* by normal humans.

To make that possible, we have to be able to change the meaning of operations (such as `$cpop * $cpop_incr`, or `$cgdp_pc - $ugdp_pc`, or `abs($sdiff)`) on objects of a given class. Fortunately, Perl provides a simple mechanism to do exactly that.

Changing the way a Perl's in-built operators behave when applied to a user-defined type is known as *operator overloading*. By overloading them, operators can be given new semantics when applied to objects of a specific class. For example, given:

```

$six = Math::BigFloat->new("6");
$seven = Math::BigFloat->new("7");
$forty_two = $six * $seven;

```

to evaluate the last statement Perl might attempt to multiply the integer representations of the two references stored in `$six` and `$seven` (i.e. the internal memory addresses of the two `Math::BigFloat` objects). That's unlikely to produce the desired result.

However, by overloading the multiplication operator, we could arrange for the multiplication of any two `Math::BigFloat` objects to produce a new `Math::BigFloat` object containing the correct value²⁵.

²⁵ The `Math::BigFloat` module actually *does* overload the basic arithmetic operators in this way, so operations on `Math::BigFloat` objects *do* work as expected.

Perl's operator overloading mechanism

Ilya Zakharevich's `overload.pm` module, which comes with the standard Perl distribution, provides access to Perl's built-in mechanism for overloading operators. To overload operators for a given class, you use the module, passing the `use` statement a list of operator/implementation pairs:

```
package Math::BigFloat;

use overload "*" => \&fmul,
             "+" => "fadd",
             "neg" => sub { Math::BigInt->new($_[0]->fneg()) };
```

Each pair consists of a keyword (which specifies the operator that is to be overloaded) and a subroutine reference (which specifies a subroutine that is to be performed when the specified operator is encountered).

The keyword must be one from the list shown in Table 1. These are the only operators that may be overloaded. Note that simple assignment isn't one of them.

Category	Operators/ Keywords	Notes
Arithmetic	"+" "-" "*" "/" "%*" "x" "." "neg"	"neg" implements unary negation. There is no overloading for unary identity (i.e. +\$obj).
Bitwise	"<<" ">>" "&" " " "^" "~"	"^" is bitwise exclusive OR, not exponentiation.
Assignment	"+=" "--=" "*=" " "/=" "%=" "**=" " "<<=" ">=" "x=" " ".=" " "++" "--"	"++" and "--" are mutators and their handler is expected to actually change the value of its first argument (e.g. \$_[0]->{val}++ for "+"). Handlers for other assignment operators may alter the first argument, but there's little point since the argument is always assigned the return value.
Comparison	"<" "<=" ">" ">=" " "==" "!=" "<=" " "lt" "le" "gt" " "ge" "eq" "ne" " "cmp"	All other operators may be automatically generated from the "<=" and "cmp" operators.
Built-in functions	"atan2" "cos" "sin" "exp" "abs" "log" "sqrt"	These override the equivalent built-in function for a specific class only.
Conversions	q{" " } "0+" "bool"	Automatically called when the context requires a string, number, or boolean.
Pseudo-operators	"nomethod" "fallback" "="	"=" does <i>not</i> overload the assignment operation.

Table 1: Overloadable operators in Perl

The subroutine reference may either be a reference to a named subroutine, or a symbolic reference (i.e. the name of the subroutine), or a reference to an anonymous subroutine. Note that all three alternatives may be used in the same `use overload` statement.

The implementation subroutine is called any time a reference to an object of the corresponding class (in the above examples, `Math::BigFloat`) is an operand of corresponding operator. If the operation was specified as a subroutine reference, then it's called as a non-method subroutine. If it is specified as a symbolic reference (i.e. a name), then it is called as a method.

In other words, if `$six` and `$seven` store `Math::BigFloat` objects, and multiplication, addition and negation are overloaded as shown above, the following series of operations:

```
$six * $seven;
$six + $seven;
-$six;
```

is automatically translated to:

```
Math::BigFloat::fmul($six,$seven,"");
$six->fadd($seven,"");
(sub { Math::BigInt->new($_[0]->fneg()) }->($six,undef,"");
```

Regardless of how it is invoked, each implementation subroutine is called with three arguments:

- The first operand of the operation,
- The second operand of the operation (or `undef` if the operation is unary, as for `fneg` above),
- A flag indicating whether the operands were reversed.

The flag is needed because—as in all other object methods—the first argument must be a reference to an object of the appropriate class (i.e. in this case a reference to a `Math::BigFloat`). If Perl detects an operation such as:

```
6 + $seven;
```

it obviously can't translate that to:

```
6->fadd($seven,""); # huh?
```

so it translates it to:

```
$seven->fadd(6,1);
```

and sets the third argument to a true value, to indicate that the arguments had to be reversed. Notice that in all the earlier examples, this argument was an empty string (i.e. false), since the first operand was always a `Math::BigNum` and so the arguments didn't need to be reversed.

Hence, for operations where the order of the operands matters (e.g. for subtraction or division), it's common to see implementation subroutines implemented like this:

```
sub subtract
{
    my ($op1, $op2, $reversed) = @_;
    ($op2,$op1) = ($op1,$op2) if $reversed;
    # then perform $op1 - $op2 as appropriate
}
```

```
use overload "-" => \&subtract;
```

It's important to bear in mind that problems may arise if both operands of a binary operator are references to objects. If the object's classes have both overloaded the operator, it's the overloading in the *first* operand's class that is invoked. In such cases it may be necessary to resort to multiple dispatch techniques to ensure that the appropriate subroutine is invoked regardless of the order of operands.

Of course, just because the operator overloading facility is sometimes useful, doesn't mean it should always be used...

The use and abuse of operators

The operator overloading mechanism gives you a powerful mechanism with which you can alter the normal behaviour of expressions and conversions involving any new class you create. The question is: *should* you?

Clearly, there are cases when overloading certain operations makes perfect sense, especially if you're creating a class to represent something that has a well-defined algebra (such as large integers, or complex numbers, or vectors, or matrices). As illustrated by the GDP calculation at the start of this section, an appropriate set of operators can greatly improve the usability of such classes and the readability of the code that uses them.

Likewise, it's often useful to overload the stringification conversion for a class. For example:

```
package CD::Music;

use overload

    q{" " } => sub { qq("${_[0]->{name}" by $_[0]->{artist}) };
```

That way, if someone writes:

```
my $cd = CD::Music->new(@messiah_data);

# and later...

print $cd, "\n";
```

they get something useful like:

"Messiah" by George Frideric Handel

rather than something merely accurate like:

CD::Music=HASH(0x1001c3e0)

Then there are cases where the "appropriateness" of specific overloadings is less certain. Consider a class implementing three-dimensional vectors, such as might be found in a graphics package:

```

$view_dir = Vector->new(x=>100,y=>0,z=>-50);
$move_dir = Vector->new(x=>20,y=>20,z=>0);

$motion_normal = $view_dir x $move_dir;
$motion_angle = $view_dir * $move_dir;

```

Clearly for such a class "x" is an appropriate operator for the cross-product operation, but is "*" the right choice for the dot-product? It certainly conveys the multiplicative nature of the operation, and it looks somewhat like a dot (albeit a big, hairy one), but would the actual "dot" operator (i.e. ".") be a better choice? That operator already has a strong association with concatenation in regular Perl, but a vector algebraist might well find:

```

$motion_angle = $view_dir . $move_dir;

```

more "intuitive". Or perhaps not. It certainly *won't* be intuitive when the "dot" operator is called to concatenate two string-interpolated variables (e.g. "\$vector1\$vector2") and produces their dot-product instead. And what would such mathematicians then make of the "^" operator? Would they expect:

```

$unit_view_dir = ^$view_dir;

```

to create a unit vector? How will they react when it produces a syntax error instead?

The problem is that, having provided some overloaded operators for a class, we have also set up an expectation in the mind of the users of that class. Because objects of the class can *sometimes* be used in the "natural" way, clients may expect that such objects can *always* be used that way. That will probably not be the case, so it's vital to document where the metaphor breaks down.

Sometimes operator overloading is used purely to provide coding expediency, rather than to promote clarity. For example, suppose we created a database class that was entirely operator-driven:

```

use OpDBM;

my $db = OpDBM->new($dbfile);

$db += [item=>"oatmeal", category=>"breakfast"];
$db += [item=>"marmalade", category=>"breakfast"];
$db += [item=>"potatoes", category=>"dinner"];

my $relation = sqrt( ~$db * [category=>"breakfast"] );

print "$_->{item}\n" while ($relation++);

```

The dozen or so people on the planet who are well-versed in abstract database theory, may rejoice in the decision to overload "+=" to handle insertions, "~" for database normalization, "*" to perform selections, "++" to iterate a set of records, and sqrt to implement the "Sort, Quashing Repeated Tuples" operation:

```

package OpDBM;
use overload
    '+' => sub { $_[0]->insert_tuple( @{$_[1]} ) },
    '~' => sub { $_[0]->normalize() },
    '*' => sub { if ($_[2]) { $_[1]->insert_tuple( @{$_[0]} ) }
                else { $_[0]->insert_tuple( @{$_[1]} ) } };

package OpDBM::Relation;
use overload
    '++' => sub { $_[0]->next() },
    'sqrt' => sub { $_[0]->sort_quash_reps() };

```

To database theoreticians, the resulting code may be obvious, intuitive, and even elegant. But it's also likely to make the task of developing and maintaining such code very much harder for the rest of us.

So when should you overload? Clearly it's a least partially a "religious issue", so there's no correct answer²⁶. As a general rule of thumb, overloading the algebraic operators for a class is appropriate when the same operators are used in the normal (non-programming) notation for whatever the class represents. Numerical and other mathematical classes are the usual candidates here.

Overloading conversion operations is also often a good idea, as it allows objects to act in standard ways the programmers may expect, especially when they are stringified. Comparison operators are another reasonable candidate for overloading in most cases, provided the values represented by objects have an obvious ordering.

A few operators are sufficiently generalized in their meaning (e.g. "+" meaning "add to") that they can reasonably be overloaded in almost any suitable context. Code like:

```
$db += [item=>"oatmeal", category=>"breakfast"];
```

or:

```
$dictionary += "algebraist";
```

or:

```
$process_group += $new_process;
```

will probably be readily intelligible to almost any programmer.

Apart from that, it's probably best to use operator overloading sparingly, the way an expert chef uses salt. Without it, your creations may be bland and indigestible, but using it indiscriminately or too often will only raise your clients' blood pressure.

Encapsulation is one of the cornerstones of object orientation, but it's the area in which Perl's support for object-oriented programming is weakest. Many would argue that enforced encapsulation is against Perl's philosophy of freedom and flexibility in programming. But there are situations when too much freedom becomes a trap, and too much flexibility makes it hard to build solid code...

The perils of trust

In practice, the lack of an in-built encapsulation mechanism rarely seems to be a problem in Perl. Most Perl programmers build classes out of standard hashes, and they and the users of their classes get by quite happily with the principle of "encapsulation by good manners". The lack of formal encapsulation doesn't matter because everybody plays nicely, keeps off the grass, and respects the "official" interface of objects. Those who don't play by the rules, and directly access a method or attribute that is supposed to be private, get what they deserve (i.e. either better performance or a nasty surprise).

²⁶ Or perhaps: '...so there are nothing but "correct" answers, most of which are mutually exclusive'.

The only problem is, this convivial arrangement doesn't scale very well. Leaving your front door open may be fine in a small town, but it's madness in the big city. Likewise, informal mechanisms that are suitable for a few hundred lines of code written by a single programmer, don't work nearly as well when the code is tens of thousands of lines long and developed by a team. Even if you could trust the entire team to maintain sufficient programming discipline to respect the notional encapsulation of attributes (a dubious proposition), accidents and mistakes happen, especially in rarely-used parts of the system that only ever get used when demonstrating to important clients.

Moreover, deliberate decisions to circumvent the rules (usually taken in the heat of hacking, out of laziness, or for efficiency) are often inadequately documented, leading to problems later in the development cycle. For example, consider a (notionally) "private" attribute of an object, which for efficiency reasons is accessed directly in an obscure part of a system. If the implementation of the object's class changes, that attribute may cease to exist.

In a more static language, this would generate an error message to be generated when next the external code attempts to access the now non-existent attribute. However, Perl's autovivification of hash entries may well "resurrect" the former attribute whenever it is accessed, so the now-incorrect access proceeds silently. Bugs such as this can be painfully difficult to diagnose and track down, especially if the original programmer has moved on by the time the problem is discovered.

Fortunately Perl's very flexibility can be turned against itself, to provide several different ways of building objects that do respect the encapsulation imposed by their classes. Here's one approach that also demonstrates an interesting application of scalar-based objects...

Encapsulation via scalars

An interesting, but less well-known approach to encapsulation uses scalar-based objects to implement a technique called the *flyweight pattern*. In the flyweight pattern, objects don't carry around their own information, so that information can't be accessed directly via the object. Instead, flyweight objects merely serve as an index into a shared table of values, stored within the class itself. For example, an object might be an integer that indexes into some table of values stored as a class attribute.

Flyweight objects are most frequently used in object-oriented languages that pass objects around by value, because flyweight objects remain extremely small (no matter how much data they "contain"), and hence are cheap to pass around. Because Perl objects are invariably accessed via references, this advantage is not significant.

However, the flyweight pattern still has something to offer in Perl, because it provides a simple mechanism for preventing direct access to object attributes, thereby enforcing encapsulation. As a bonus, it also provides a means of easily keeping track of every object in a class.

Name, rank, and serial number

Figure 9 shows a flyweight implementation of the Soldier class. The entire class is contained in a pair of curly braces, to ensure that any lexical variables declared within their scope are

not directly accessible outside that scope. Not surprisingly then, the first thing the class does is declare some lexical variables.

```

package Soldier;
$VERSION = 2.00;
use strict;

{
    # Table storing references to hashes containing object data
    my @_soldiers;

    # Allowable attributes and their default values
    my %_fields = (name=>'???', rank=>'???', serial_num=>-1);

    # Constructor adds object data to table and blessed a scalar
    # storing the index of that data

    sub new
    {
        my ($class, %args) = @_ ;
        my $dataref = { %_fields };
        foreach my $field ( keys %_fields )
        {
            $dataref->{$field} = $args{$field}
                if defined $args{$field};
        }
        push @_soldiers, $dataref;
        my $object = $#_soldiers;
        bless \$object, $class;
    }

    # These methods provide the only means of accessing object attributes
    # (note that only rank can be changed)

    sub get_name      { return @_soldiers[ ${$_[0]} ]->{name} }
    sub get_rank      { return @_soldiers[ ${$_[0]} ]->{rank} }
    sub get_serial_num { return @_soldiers[ ${$_[0]} ]->{serial_num} }

    sub set_rank
    {
        my ($indexref, $newrank) = @_ ;
        @_soldiers[ ${$indexref} ]->{rank} = $newrank
    }

    # This class method provides an iterator over every object

    my $_cursor = -1;
    sub each
    {
        my $nextindex = ++$_cursor;
        if ($nextindex < @_soldiers)
        {
            return bless \$nextindex, ref($_[0])||$_[0];
        }
        else
        {
            $_cursor = -1;
            return undef;
        }
    }
}

```

Figure 9: The Soldier class implemented via scalars

The lexical array `@_soldiers` is used to store the data for each object. That data is directly accessible to the methods declared within the curly braces, but nowhere else. This restriction will eventually provide the desired encapsulation of object data.

The lexical hash `%_fields` performs the dual function of recording (in its keys) the names of valid attributes of a `Soldier` object, and storing (in its values) the default values for those attributes.

The constructor begins like most others we've seen so far, by creating a new anonymous hash and initializing it with the default attribute values for the class. It then loops over the valid fields of the class, overwriting those default values with any corresponding argument that was passed to the constructor.

At this point a typical constructor would bless and return the reference in `$objref`, thereby making the anonymous hash into the new object. Instead, `Soldier::new` pushes the hash onto the end of the encapsulated `@_soldiers` array, and blesses a scalar storing the index of that newly added array element instead.

Thus a constructor call such as:

```
my $grunt = Soldier->new(name => "Smith, J.",
                        rank => "private",
                        serial_num => 149162536);
```

leaves `$grunt` with a reference to a scalar (i.e. to the index of the data), rather than a reference to a hash (i.e. to the data itself). Figure 10 illustrates the process.

Theoretically, the effect is the same, since we have the index and know which array it refers to, so we can still find the actual data. In practice, however, there's an important difference. Outside the curly braces surrounding the class, the `@_soldiers` array is inaccessible, so even though we have the index for the object's data, we can't access that data directly.

Controlled access

Instead, it's up to the accessor methods of the class to provide the required access. Since they are all defined within the encapsulating curly braces, they *do* have access to `@_soldiers`, and so they can dereference the blessed index (i.e. `$_[0]`), index into the array to get a reference to the appropriate hash data (`$_soldiers[$_][1]`), and then access the correct field of that data using the arrow notation (`$_soldiers[$_]->{name}`).

Note that the implementation shown in Figure 9 doesn't provide "write" accessors for a `Soldier`'s name or serial number. This provides real data security, since without the accessors there is no way of modifying these attributes, once they are set.

We could try to impose a new method on the class:

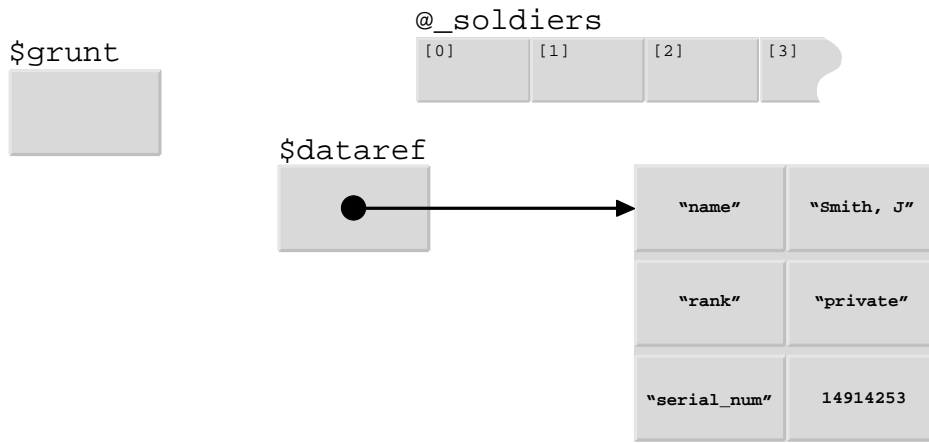
```
package main;
use Soldiers;

my $general = Soldier->new( name => "Caesar, G.J.",
                          rank => "Prodictator",
                          serial_num => "MMXLVIII");

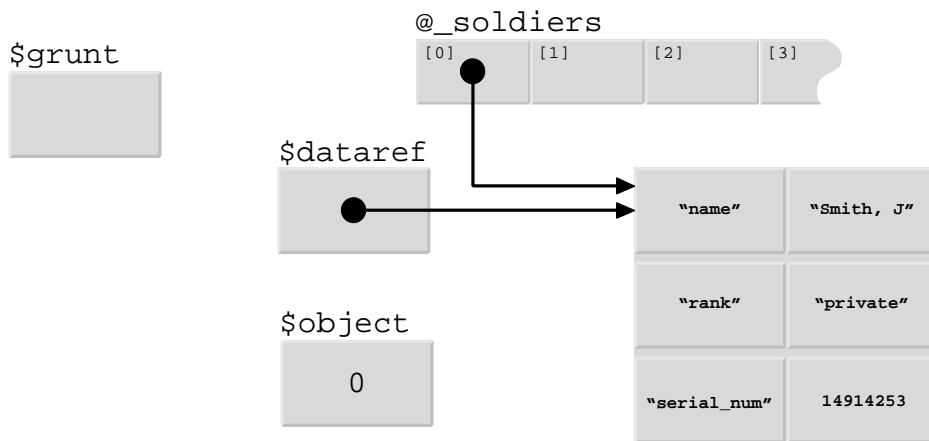
# Oops, that serial number was out by one.
# Strange, there's no method to change it.
# Oh well, let's just add one ourselves...
Figure sub Soldiers::set_serial_num { $_soldiers[$_]->{serial_num} = $_[1] }

# ...and use it...

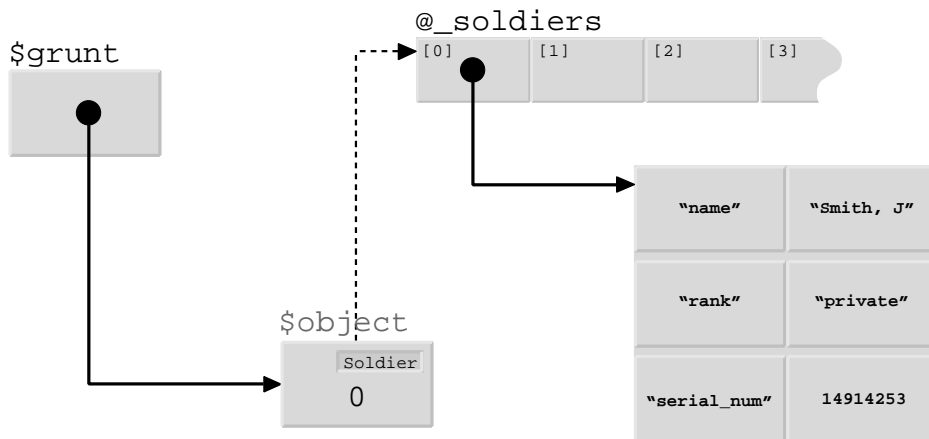
$general->set_serial_num("MMXLIX");
```



i. After my \$dataref = {%_fields};
and foreach my \$field...



ii. After push @_soldiers, \$dataref;
and my \$object = \$#_soldiers;



iii. After bless \\$object, \$class;
and constructor returns

Figure 10: Construction of a Soldier object

but that wouldn't circumvent encapsulation. Although the new method *will* be in the class's namespace (and hence, callable through its objects), it *won't* be in the lexical scope of the original encapsulating curly braces, so it won't have access to the lexical `@_soldiers` array.

It's worth noting that Perl will visit a satisfying form of Instant Justice on the author of this code. Since the code doesn't use `strict`, Perl will conclude that the `@_soldiers` array being modified in `Soldier::set_serial_num` is the package variable `@main::_soldiers`. Thus, the code will execute without complaint, yet mysteriously fail to update any soldier's serial number, leading to happy hours of fruitless debugging.

Roll call

The other advantage of a scalar-based object representation like this is that the class itself has direct and continuing access to the data of every object blessed into it. That makes it very easy to provide class methods to iterate that data.

The `Soldier` class demonstrates this by providing an iterator method (`Soldier::each`), which steps through the indices of the `@_soldiers` array, returning a blessed version of each index (i.e. a `Soldier` object). The method can be used like this:

```
while (my $soldier = Soldier->each)
{
    printf "name: %s\nrank: %s\n s/n: %d\n\n",
        $soldier->get_name(),
        $soldier->get_rank(),
        $soldier->get_serial_num();
}
```

By the way, as elegant as it might look, don't be tempted to write:

```
while (my $soldier = each Soldier) {...}
```

hoping that this is one of the few places where the "indirect object" syntax will work. It isn't. Instead, Perl will assume you wanted to use the built-in `each` function to iterate the package hash `%Soldier`, and just forgot the `"%"` prefix. Once again, use `strict` will prevent Perl from helping you cut your own throat like this.

Generic programming is an important tool in the object-oriented armory. But Perl's support for closures, dynamic typing of variables, and run-time code evaluation makes explicit generic mechanisms largely irrelevant...

Why Perl doesn't need special generic mechanisms

The ability to specify generic code structures, independent of the details of class type, is an important component of most object-oriented languages. Anyone who has ever had to rewrite the `List` class to cater for various different types of elements (`List_of_CDs`, `List_of_Soldiers`, `List_of_Bugs`, `List_of_List_of_BigInts`, etc.) knows that generic types and subroutines are a powerful means of reducing code duplication, and simplifying maintenance. In fact, some would contend that genericity is more important than hierarchical abstraction in this respect.

Surprisingly, then, Perl offers no explicit built-in mechanism for creating generic classes or generic subroutines²⁷. There are no "templates", no "metaclasses", no "parametric types", no "class generators". The reason is simple: they aren't needed.

Explicit mechanisms for specifying generic structures are usually found in languages with static typing (i.e. where the type of each object is determined—and fixed—during compilation). Such languages need a way of separating the generic *form* of a data structure from the type-specific implementation of that data structure, for a given set of classes (usually called its *type parameters*).

In other words, such languages need a way of indicating a place-holder for a type, so that it's possible to specify something general like:

"Let L be a list of objects of type <whatever>. To insert a new element E (also of type <whatever>), for each object O already in the L, use a subroutine S to compare O and E. The first time the comparison is false, insert E before O in the list, and set the current element pointer to the newly inserted element."

Then the compiler can fill in the place-holders with actual types, usually with the assistance of some additional syntax to bind specified types to the place-holder <whatever>.

Perl's type system is completely different from this. Perl objects don't have statically-associated class types; objects can be reblessed at any time. Perl variables don't have static types either; they can store objects blessed into any class (or into no class at all). So in Perl, we can get away with specifying a generic list insertion as follows:

"Let L be a list of objects (of any type, as usual). To insert a new element E (of any type), for each object O already in the L, use a subroutine S to compare O and E. The first time the comparison is false, insert E before O in the list, and set the current element pointer to the newly inserted element."

Or, in actual Perl code:

```
sub List::insert
{
    my ($L, $E, $S) = @_;
    my $index;
    for ($index=0; $index < @{$L->{elements}}; $index++)
    {
        my $O = $L->{elements}->[$index];
        last unless $S->($O, $E);
    }
    splice @{$L->{elements}}, $index, 0, $E;
    $L->{current} = $index;
}
```

The dynamic typing of Perl variables means that there is no need for place-holders, since any scalar variable will happily hold a reference to an object of any class, and arguments of any type(s) may be passed to any subroutine.

Of course, there's a price to pay. The earlier statically-typed version has access to more information (namely the type of object the list is supposed to store). With this information, it

²⁷ ...except, perhaps, the `AUTOLOAD` mechanism, which could be viewed as a run-time generic method that reproduces the functionality, and to some extent the internal structure, of the specific methods it replaces.

can pick up errors at compile-time (e.g. attempting to insert an object E of the wrong type, or using the wrong comparison subroutine). In the Perl version, these problems would not surface until the code is actually executed (and perhaps not even then).

Another reason why Perl has no need for explicit genericity is Perl's excellent range of built-in datatypes. As the above code example illustrates, Perl's powerful arrays, hashes, references to data and subroutines, etc., are already generic, in that they are capable of storing (or referring to) any Perl data-type. Thus, when implementing the insertion subroutine for our generic list class, we can just use a regular Perl array as the basis of the class, confident that it will be able to store whatever kind of objects we choose to store.

Of course, there's always more than one way to do things in Perl, so it's not hard to whip up a proper generic class when you need it...

Using specific mechanisms anyway

Yet another reason why Perl doesn't need explicit genericity is that it provides other powerful mechanisms that can achieve the same effects. Closures, for example, are a general means of binding specific data (and hence data-types) into a generic subroutine. The `eval` function provides an even more powerful code-generation mechanism, allowing us to build an entire class, using interpolated variables in a string as place-holders. This section explains both of these techniques.

Closures as generic methods

One problem with the generic `List::insert` code shown above is that the user has to pass a reference to some comparison subroutine (which is accessed via `$S`) each time they wish to do an insertion. That's tedious and unnecessary.

A better solution would be to provide a method that *generates* variants of the `List::insert` method, with the appropriate comparison subroutine "hard-wired" in. For example, we could write:

```
sub List::generate_insert
{
  my ($class, $S) = @_;
  return sub
  {
    my ($L, $E) = @_;
    my $index;
    for ($index=0; $index < @{$L->{elements}}; $index++)
    {
      my $O = $L->{elements}[$index];
      last unless $S->($O, $E);
    }
    splice @{$L->{elements}}, $index, 0, $E;
  }
}
```

`List::generate_insert` takes a single argument, which is a reference to a subroutine. It then creates and returns a new anonymous subroutine that is functionally

identical to `List::insert`, except that it no longer requires a third (`$$`) argument. Instead, the lexical variable `$$`, which was created in the first line of `List::generate_insert`, is used. It is still in scope when the anonymous subroutine is created, so it's okay to use it within that subroutine. And because the subroutine uses it, it survives the end of the call to `List::generate_insert`.

Of course, to make it usable, we still have to associate the newly created anonymous subroutine with the `insert` method of a specific class. If, for example, we wanted `List_of_Employee::insert` to use the anonymous subroutine we could simply assign the result of the call to `List::generate_insert` to the appropriate typeglob:

```
*List_of_Employee::insert = List->generate_insert(\&List_of_Employee::_compare);
```

We passed `List::generate_insert` a reference to the subroutine `List_of_Employee::_compare`, so that the new class's `insert` method used its own `_compare`. This approach gives us some flexibility in deciding which comparison subroutine is used by which `insert`. For instance we might want to ensure the same sorting order for both general employees and management level staff:

```
*List_of_Manager::insert = List->generate_insert(\&List_of_Employee::_compare);
```

On the other hand, this might be a coding error (and a difficult one to detect, at that). If each specific list class's `insert` should *always* use the same class's `_compare`, we could rewrite the `List::generate_insert` method to produce a safer generator method:

```
sub List::generate_insert_for
{
  my ($class, $newclass) = @_;
  no strict "refs";
  my $$ = \&{"${newclass}::_compare"};
  *{"${newclass}::insert"} = sub
  {
    my ($L, $E) = @_;
    my $index;
    for ($index=0; $index < @{$L->{elements}}; $index++)
    {
      my $O = $L->{elements}[$index];
      last unless $$->($O, $E);
    }
    splice @{$L->{elements}}, $index, 0, $E;
  }
}
```

Instead of a reference to the comparison function, this version takes the name of the new class as its argument. It then uses that class name to build the name of the appropriate comparison subroutine (i.e. `"${newclass}::_compare"`). That name is used as a symbolic reference to the subroutine itself (`&{"${newclass}::_compare"}`), which is converted via a leading backslash (`\&{"${newclass}::_compare"}`) to a normal reference. The anonymous subroutine is then constructed exactly as before, but now it is automatically assigned it to the correct typeglob, which is also accessed via a symbolic reference (`*{"${newclass}::insert"}`).

So now, to create a suitable `List_of_Manager::insert` method, we write:

```
List->generate_insert_for("List_of_Employee");
```

This same approach could easily be extended to generate the other necessary methods for a list class:

```
List->generate_first_for("List_of_Employee");
List->generate_next_for("List_of_Employee");
List->generate_insert_for("List_of_Employee");
List->generate_delete_for("List_of_Employee");
```

and, better still, aggregated into a single subroutine:

```
sub List::generate_methods_for
{
    my ($class, $newclass) = @_;
    List->generate_first_for($newclass);
    List->generate_next_for($newclass);
    List->generate_insert_for($newclass);
    List->generate_delete_for($newclass);
}
```

Even better, that subroutine could be called `List::import` instead of `List::generate_methods_for`. Remember that `import` is automatically called whenever a module is use'd, so we could put the various generic method generators of class `List` into a `List.pm` module, and then create list classes whenever they are needed like so:

```
use List "List_of_Employee";
use List "List_of_Tasks";
use List "ClientList";
# etc.
```

Of course, we'd need to declare the various `_compare` subroutines in `BEGIN` blocks to makes sure each is available when each new list type is use'd into existence during compilation:

```
BEGIN
{
    sub List_of_Employee::_compare
    {
        return $_[0]->{ID} < $_[1]->{ID}
    }
}

use List "List_of_Employee";
```

Otherwise, there would be no way to create the necessary reference to it within `List::generate_insert_for`.

Alternatively, we could go back to passing the comparison subroutine explicitly (as a second argument to `List::generate_insert_for`):

```
sub List::generate_insert_for
{
    my ($class, $newclass, $S) = @_;
    no strict "refs";
    *{"${newclass}::insert"} = sub
    {
        my ($L, $E) = @_;
        my $index;
        for ($index=0; $index < @{$L->{elements}}; $index++)
        {
            my $O = $L->{elements}[$index];
            last unless $S->($O, $E);
        }
        splice @{$L->{elements}}, $index, 0, $E;
    }
}
```

modify `List::import` accordingly:

```

sub List::import
{
    my ($class, $newclass, $comparison_sub) = @_;
    List->generate_first_for($newclass);
    List->generate_next_for($newclass);
    List->generate_insert_for($newclass,$comparison_sub);
    List->generate_delete_for($newclass);
}

```

and then create entire classes in a single line:

```

use List "List_Employee", sub {$_[0]->{ID} < $_[1]->{ID}};
use List "Manager::List", sub {$_[0]->{revenue} > $_[1]->{revenue}};

# and later...

my $minions = List_Employee->new();
my $masters = Manager::List->new();

```

Figure 11 shows the complete generic List class, as it would be specified in the List.pm module.

The same approach could be adapted to any generic class that required specific subroutine references, or class name strings, or other class-specific data to be interpolated into some place-holder within the generic code. The values to be interpolated would be passed as arguments to use, and would be distributed to the appropriate generator subroutines by the generic class's import method.

eval'ing generic classes

There are two main drawbacks to the approach described in the previous section. Firstly, invoking the comparison subroutine through a reference (i.e. \$S) is relatively slow, which is unfortunate in a fundamental data structure like a list. Secondly, the use of the various generate_... methods, each with an embedded anonymous subroutine representing a single generic method, fragments the generic class and makes it more difficult to understand and maintain.

Both problems can be overcome at once by changing the way the individual subroutines are generated. Instead of creating an anonymous subroutine and assigning it into a typeglob, we could use an eval to convert a block of text (containing the specification of the generic class) into actual Perl code. Figure 12 shows the generic List class from Figure 11 modified in this way.

The _replace subroutine takes a reference to an anonymous hash as its first argument and a string as its second. The hash specifies a series of substitutions to perform on the string. Each key of the hash is a literal pattern, each value is the corresponding replacement text.

The List::import subroutine does effectively the same job as the previous version, but in a very different way. It first extracts the name of the new class and the code for that class's _compare subroutine from the argument list.

Then, if necessary, it reads in the "template" for the List class that is specified after the __DATA__ token, and stores it in the package lexical variable \$code. This read will only be performed the very first time a use List is issued in a program, after which the text is cached in \$code.

```

package List;
$VERSION = 1.00;

sub List::import {
    my ($class, $newclass, $comparison_sub) = @_;
    List->generate_new_for($newclass);
    List->generate_first_for($newclass);
    List->generate_next_for($newclass);
    List->generate_insert_for($newclass,$comparison_sub);
    List->generate_delete_for($newclass);
}

sub List::generate_new_for {
    my ($class, $newclass) = @_;
    *{"${newclass}::new"} = sub {
        my ($class, @data) = @_;
        bless { current => 0, data =>      [@data], }, ref($class)||$class;
    }
}

sub List::generate_first_for {
    my ($class, $newclass) = @_;
    *{"${newclass}::first"} = sub {
        my ($self) = @_;
        return \${self->{data}}[${self->{current}}]
            if (\${self->{current}}=0) < @{\${self->{data}}};
    }
}

sub List::generate_next_for {
    my ($class, $newclass) = @_;
    *{"${newclass}::next"} = sub {
        my ($self) = @_;
        return \${self->{data}}[${self->{current}}]
            if (++\${self->{current}}) < @{\${self->{data}}};
    }
}

sub List::generate_insert_for {
    my ($class, $newclass, $compare) = @_;
    *{"${newclass}::insert"} = sub {
        my ($self, $newelem) = @_;
        my $index;
        for ($index=0; $index < @{\${self->{data}}}; $index++) {
            my $nextelem = \${self->{data}}[$index];
            last unless $compare->($nextelem, $newelem);
        }
        splice @{\${self->{data}}}, $index, 0, $newelem;
        \${self->{current}} = $index;
    }
}

sub List::generate_delete_for {
    my ($class, $newclass) = @_;
    *{"${newclass}::delete"} = sub {
        my ($self) = @_;
        splice @{\${self->{data}}}, \${self->{current}}, 1;
    }
}

1; # to ensure the "use" succeeds

```

Figure 11: The List module

```

package List;
$VERSION = 2.00;
use strict;

sub _replace {
    my ($substitute, $text) = @_;
    $text =~ s/\Q$_/$substitute->{$_}/g foreach (keys %$substitute);
    return $text;
}

my $code;
sub import {
    my ($class, $newclass, $compare) = @_;
    unless (defined $code) { local $/; $code = <DATA> }
    eval _replace {'<<CLASS>>'=>$newclass, '<<COMPARE>>'=>$compare} => $code;
}

1; # to ensure the "use" succeeds

__DATA__
package <<CLASS>>;

sub new {
    my ($class, @data) = @_;
    bless { current => 0, data => [@data] }, ref($class)||$class;
}

sub compare { <<COMPARE>> }

sub first {
    my ($self) = @_;
    return \($self->{data}[$self->{current}])
        if ($self->{current}=0) < @{$self->{data}};
}

sub next {
    my ($self) = @_;
    return \($self->{data}[$self->{current}])
        if (++$self->{current}) < @{$self->{data}};
}

sub insert {
    my ($self, $newelem) = @_;
    my $index;
    for ($index=0; $index < @{$self->{data}}; $index++) {
        my $nextelem = $self->{data}[$index];
        last unless compare($nextelem, $newelem);
    }
    splice @{$self->{data}}, $index, 0, $newelem;
    $self->{current} = $index;
}

sub delete {
    my ($self) = @_;
    splice @{$self->{data}}, $self->{current}, 1;
}

```

Figure 12: The List module using eval

To generate the specific code needed we could also have used a "here document", or an interpolation module such as `Interpolation` or `Text::Template`²⁸, or even the standard `sprintf` function. However reading from `<DATA>` allows us to cleanly distinguish between the actual code implementing the `List.pm` module (everything before `__DATA__`), and the code template it uses (everything after `__DATA__`).

The template code is passed through `_replace` to put the new class's name and the comparison code into the appropriate place-holders. Once these slots have been filled in, the (now-specific) code is eval'ed into the current program.

The remainder of the file consists of the code template for the generic list class. Note that the entire class is laid out together—with the place-holders `<<CLASS>>` and `<<COMPARE>>` for the generic bits—making the structure, function, and level of genericity of the class much clearer.

This version of the `List` class is used in much the same "one-line" manner as the previous one—the only difference is that the comparison code is now specified as a string (in single quotes to avoid the embarrassment of premature interpolation):

```
use List "List_Employee", '$_[0]->{ID} < $_[1]->{ID}';
use List "Manager::List", '$_[0]->{revenue} > $_[1]->{revenue}';

# and later...

my $minions = List_Employee->new();
my $masters = Manager::List->new();
```

Once again, the technique can be easily adapted to generate generic classes for any purpose and with any number of place-holders.

Many people come to Perl on the rebound from a bad experience with some other object-oriented language. With a little guidance, the transition can be surprisingly painless. Here, for example, is object-oriented Perl from a C++ perspective...

Perl and C++

For better or worse, to much of the programming world C++ is object-oriented programming. Devised by Bjarne Stroustrup at the Bell Research Laboratories in the early 80's as an object-oriented successor to the C programming language, C++ has undergone perhaps the most public and collaborative evolution of any programming language, culminating in the recent ANSI C++ standard.

Like Perl, C++ is a hybrid language with object-oriented features layered over an original imperative language. Syntactically and semantically, its non-object-oriented components are almost completely backwards compatible with the C programming language, whilst its object-oriented features draw most heavily on Simula.

²⁸ ...both by Mark-Jason Dominus, and available from the CPAN...

Objects

C++ objects are structured regions of memory that store one or more typed data members. In other words, every object is a record of various fields. In Perl, objects may also be record-like structures (i.e. hashes), but they may also be arrays, scalar variables, subroutine references, or a any other Perl datatype.

C++ objects may be stored directly in statically-typed variables, or dynamically created and accessed via typed pointers or references. Perl objects may similarly be variables or unnamed values, and are always accessed via references²⁹ stored in dynamically-typed variables. Unlike C++, in Perl there is no need for a manual deallocation mechanism like `delete`, since all objects in Perl are automatically garbage-collected.

C++ also permits the definition of "static" data members that are (conceptually) shared by all objects of a given class. Perl has no equivalent construct, but it is easy to set up such shared attributes using lexical variables of appropriately restricted scope.

C++ objects are created either by static declaration or by dynamic allocation using the `new` operator. Perl objects are almost always created dynamically, in a method that is often called `new`.

Classes

A class in C++ is a specification of the data and function members (i.e. methods) possessed by a particular kind of object. Classes in Perl also define the methods of a type of object, but do not normally directly specify the attributes possessed by such objects³⁰. Attribute specification is typically arranged by the constructor method (e.g. `new`).

In C++, a class specifies a local namespace in which data and function members exist, but C++ also has a separate higher-level `namespace` mechanism with which two or more classes can be grouped. Perl's `package` construct does double-duty as both a namespace and a class specification mechanism, so there is no such ability to construct hierarchical namespaces.

Perl provides better resources for run-time type information than does C++. Whereas a C++ program is restricted to the data provided by the standard `typeid` function, and limited to using dynamic casts to verify class compatibility, Perl allows almost every aspect of a class's structure and capabilities to be interrogated at run-time: class name via the `ref` function, hierarchical relationships via the `UNIVERSAL::isa` subroutine, and method compatibility via the `UNIVERSAL::can` subroutine.

Perl does not directly support generic classes such as those provided by C++ templates. In practice this presents very few problems because the combination of Perl's closure mechanism, interface polymorphism, and dynamic-typing makes generic types largely unnecessary.

²⁹ A Perl reference is semantically closer to a C++ pointer than to a C++ reference. In Perl, references are not automatically dereferenced as they are in C++, nor must they be permanently bound to a given variable.

³⁰ Although there are various modules that make it possible to declaratively specify a class's attributes.

Methods

C++ and Perl are both hybrid languages that allow code to be executed in "stand-alone" subroutines as well as methods.

In C++, a class's function members are declared as part of the class specification and may be defined at the same point or anywhere else, provided the appropriate class qualifier is used (i.e. `ClassName::functionName`). Perl is even more liberal in this respect: a class method may be declared *and* defined anywhere, provided it is suitably qualified (using the same qualification syntax as C++).

Every C++ member function has a specific signature determined by its name and the number and types of arguments it takes. C++ methods may be overloaded, may have default argument values, and may also pass arbitrary arguments (using the "..." specifier). Perl methods have no static checking of parameter types, and Perl unifies the many "variable argument list" mechanisms of C++ by simply passing arguments as an actual variable-length list. There is no signature-based method selection (like C++ overloading), but the effect can be achieved using multimethod techniques.

C++ member functions are called on an object or object reference using the "dot" operator (`varOrRef.method(args)`). Methods may also be invoked through a pointer using the "arrow" operator (`ptr->method(args)`). In Perl, methods are always invoked through a reference to an object, using the arrow operator (`$ref->method(args)`). Unlike C++, in Perl if the method takes no arguments, the trailing parentheses indicating a subroutine call may be omitted.

C++ allows pointers or references to member functions to be used to call those functions on specific objects using the `ptr->*funcptr()` syntax. Perl allows references to methods to be used in the same way, using the `$ref->$methodRef()` notation. Unlike C++, Perl also allows methods to be called "by name", by storing a suitable character string—rather than a reference—in the `$methodRef` variable.

In both languages a method may act like a procedure or a function, depending on whether it chooses to return a value. Both languages provide a `return` statement to specify such return values. However, unlike C++, where a member function that does not return a value must have a return type of `void`, Perl methods do not require (nor allow) any form of return-type specification.

C++ provides the special constant `this` within each member function, which is a pointer to the object on which the method was called. In Perl, a reference to the invoking object is instead passed as the first argument to the call. It is typically extracted from the argument list and stored in a variable called `$self`.

Both C++ and Perl allow class methods to be defined within a class. In C++, such member functions are defined with the `static` qualifier and are called using the syntax `ClassName::method(args)`. In Perl, such methods are defined like all other methods, and differ only in that they expect the class name—rather than an object reference—as their first argument. They are called using the syntax `ClassName->method(args)`.

Both languages also support the definition of class-specific versions of the standard set of operators (i.e. operator overloading), and as in C++, overloaded operators in Perl may either be regular subroutines or specific object methods.

Constructors and destructors

C++ classes typically provide a special member function (with the same name as the class itself) that may be used to initialize objects when they are created. Perl has no comparable in-built initialization mechanism. Instead, a regular class method (typically called `new`) is used to both create and initialize objects.

C++ also provides for destructor functions, which are automatically called on an object just before it goes out of scope or is otherwise deallocated. Perl also allows for destructor methods to be defined using the special method name `DESTROY`.

Encapsulation

Every data and function member of a C++ class has some associated "accessibility" – `public`, `protected`, or `private` – which determines the scopes from which it can be directly accessed. Perl has no equivalent concept, and does not enforce any form of encapsulation on attributes or methods of objects. There are, however, several programming techniques that permit both attributes and methods to be appropriately restricted in accessibility.

Inheritance

Both C++ and Perl support optional multiple inheritance of superclasses, but in quite different ways. In C++, the classes from which a given class inherits are determined at compile-time by the class definition. The classes that a given Perl package inherits are determined at run-time by the contents of that package's `@ISA` array.

A subclass in C++ does not have access to the private data and function members of its superclasses³¹. Because the attributes and methods of a Perl class are entirely unencapsulated, there is no equivalent restriction in Perl. Likewise Perl does not support access variations along the lines of C++'s `protected` or `private` inheritance.

Perl does not have a mechanism corresponding to virtual inheritance in C++, nor does it need one, since object attributes are determined dynamically by constructors, rather than statically by class definitions. In practice, the commonest forms of class implementation all provide implicit virtual inheritance of attributes³².

Unlike C++, Perl classes all implicitly inherit from a single common class called `UNIVERSAL`.

Polymorphism

In C++, methods are implicitly non-polymorphic unless they are specifically marked as being `virtual`. All Perl methods are implicitly polymorphic and there is no way to mark them as non-polymorphic. Unlike C++, in Perl any method may be redefined in any derived class.

³¹ ...except in the highly unusual case where the derived class is also a friend of the base class.

³² ...mainly as a consequence of the uniqueness of keys in the hashes on which most classes are based.

C++ polymorphism is controlled by class hierarchies, because virtual functions are called through typed pointers or references. In Perl, all variables are dynamically typed, and so may store a reference to any class of object at any time. Thus, Perl provides the more general form of polymorphism—interface polymorphism—in which any object (regardless of its class hierarchy membership) may respond to any method call for which it has a suitably named method.

C++ allows base class member functions to be accessed from derived class member functions, even if the derived class redefines the function in question. This access is achieved by fully qualifying the nested function call with the name of the desired ancestral class. Perl has the same mechanism. However, Perl also provides a special "pseudo-class" called SUPER that may be used to delegate a method dispatch to an unspecified ancestral class (i.e. whichever one actually provides the inherited method).

Perl has no method abstraction construct corresponding to C++'s pure virtual member function declaration. Instead, in keeping with Perl's dynamically-typed nature and run-time checking philosophy, if an abstract method is required, a normal method is specified and made to immediately throw an exception.

Comparative syntax

Table 2 shows a translation into Perl of the fundamental object-oriented features of C++.

Construct	C++	Perl
Comment	<pre>// Comment to EOL /* Delimited comment */</pre>	<pre># comment from '#' to eol</pre>
Assignment	<pre>variable = value;</pre>	<pre>\$variable = value;</pre>
Temporary variable	<pre>className variable = init;</pre>	<pre>my \$variable = init;</pre>
Class definition	<pre>class className { specification };</pre>	<pre>package className; specification</pre>
Class derivation	<pre>class subclassName : superclassName(s) { specification };</pre>	<pre>package subclassName; @ISA = qw(superclassName(s)); specification</pre>
Attribute specification	<pre>class className { type memberName; };</pre>	<pre>bless { memberName=>type->new() }, className;</pre>
Class attribute specification	<pre>class className { static type memberName; }; type className::memberName =init;</pre>	<pre>package className; { my \$var = type->new(init); sub fieldName { \$var = \$_[1] if @_>1; \$var } }</pre>
Object instantiation	<pre>ptr = new className (args);</pre>	<pre>\$ref = className->new(args);</pre>

Table 2: Selected comparative syntax for C++ and object-oriented Perl

Construct	C++	Perl
Method definition	<pre>class className { returnType methodName(args) { statements return returnValue; } }</pre>	<pre>package className; sub methodName { my @args = @_; statements; return returnValue; }</pre>
Polymorphic method definition	<pre>virtual returnType methodName(args) { statements return returnValue; }</pre>	<pre>sub methodName { my @args = @_; statements; return returnValue; }</pre>
Abstract method definition	<pre>virtual returnType methodName() = 0;</pre>	<pre>sub methodName { die "Abstract method" }</pre>
Constructor definition	<pre>className(args) { statements }</pre>	<pre>sub new { my (\$classname,@args) = @_; my \$self = bless {}, \$classname; statements; return \$self; }</pre>
Destructor definition	<pre>~className() { statements }</pre>	<pre>sub DESTROY { statements }</pre>
Method invocation	<pre>objref.methodName(args); objptr->methodName(args);</pre>	<pre>\$objref->methodName(args);</pre>
Indirect method invocation	<pre>retType (class::*methptr)(args) = class::methodName; objref.*methptr(args); objptr->*methptr(args);</pre>	<pre>\$methref = \&class::methodName; \$methsymref = "methodName"; \$ref->\$methref(args); \$ref->\$methsymref(args);</pre>
Class method invocation	<pre>className::methodName();</pre>	<pre>className->methodName();</pre>
Access to message target	<pre>this</pre>	<pre>my (\$self) = @_</pre>
Access to superclass method	<pre>this->superclass::methodName();</pre>	<pre>\$self->SUPER::methodName();</pre>
Class type identification	<pre>classDescriptor = typeid(object);</pre>	<pre>\$className = ref(\$object);</pre>
Exception handlers	<pre>try { statements } catch { handler }</pre>	<pre>unless (eval { statements; 1 }) { handler }</pre>
Raising an exception	<pre>throw exceptionType(args);</pre>	<pre>die "exceptionText";</pre>

Table 2: Selected comparative syntax for C++ and object-oriented Perl (continued)