24

Design and Programming

Keep it simple:
as simple as possible,
but no simpler.
– A. Einstein

Design and programming language — classes — inheritance — type checking — programming — what do classes represent? — class hierarchies — dependencies — containment — containment and inheritance — design tradeoffs — use relationships — programmed-in relationships — invariants — assertions — encapsulation — components — templates — interfaces and implementations — advice.

24.1 Overview [lang.overview]

This chapter considers the ways programming languages in general and C++ in particular can support design:

§24.2 The fundamental role of classes, class hierarchies, type checking, and programming itself

§24.3 Uses of classes and class hierarchies, focussing on dependencies between different parts of a program

§24.4 The notion of a *component*, which is the basic unit of design, and some practical observations about how to express interfaces

More general design issues are found in Chapter 23, and the various uses of classes are discussed in more detail in Chapter 25.

24.2 Design and Programming Language [lang.intro]

If I were to build a bridge, I would seriously consider what material to build it out of. Also, the design of the bridge would be heavily influenced by the choice of material and vice versa. Reasonable designs for stone bridges differ from reasonable designs for steel bridges, from reasonable designs for wooden bridges, etc. I would not expect to be able to select the proper material for a bridge without knowing a bit about the various materials and their uses. Naturally, you don't have to be an expert carpenter to design a wooden bridge, but you do have to know the fundamentals of wooden constructions to choose between wood and iron as the material for a bridge. Furthermore, even though you don't personally have to be an expert carpenter to design a wooden bridge, you do need quite a detailed knowledge of the properties of wood and the mores of carpenters.

The analogy is that to choose a language for some software, you need knowledge of several languages, and to design a piece of software successfully, you need a fairly detailed knowledge of the chosen implementation language – even if you never personally write a single line of that software. The good bridge designer respects the properties of materials and uses them to enhance the design. Similarly, the good software designer builds on the strengths of the implementation language and – as far as possible – avoids using it in ways that cause problems for implementers.

One might think that this sensitivity to language issues comes naturally when only a single designer/programmer is involved. However, even in such cases the programmer can be seduced into misusing the language due to inadequate experience or undue respect for styles of programming established for radically different languages. When the designer is different from the programmer – and especially if they do not share a common culture – the likelihood of introducing error, inelegance, and inefficiencies into the resulting system approaches certainty.

So what can a programming language do for a designer? It can provide features that allow the fundamental notions of the design to be represented directly in the programming language. This eases the implementation, makes it easier to maintain the correspondence between the design and the implementation, enables better communication between designers and implementers, and allows better tools to be built to support both designers and implementers.

For example, most design methods are concerned about dependencies between different parts of a program (usually to minimize them and to ensure that they are well defined and understood). A language that supports explicit interfaces between parts of a program can support such design notions. It can guarantee that only the expected dependencies actually exist. Because many dependencies are explicit in code written in such a language, tools that read a program to produce charts of dependencies can be provided. This eases the job of designers and others that need to understand the structure of a program. A programming language such as C++ can be used to decrease the gap between design and program and consequently reduce the scope for confusion and misunderstandings.

The key notion of C++ is that of a class. A C++ class is a type. Together with namespaces, classes are also a primary mechanism for information hiding. Programs can be specified in terms of user-defined types and hierarchies of such user-defined types. Both built-in and user-defined types obey statically checked type rules. Virtual functions provide a mechanism for run-time binding without breaking the static type rules. Templates support the design of parameterized types. Exceptions provide a way of making error handling more regular. These C++ features can be used without incurring overhead compared to C programs. These are the first-order properties of C++

that must be understood and considered by a designer. In addition, generally available major libraries – such as matrix libraries, database interfaces, graphical user interface libraries, and concurrency support libraries – can strongly affect design choices.

Fear of novelty sometimes leads to sub-optimal use of C++. So does misapplication of lessons from other languages, systems, and application areas. Poor design tools can also warp designs. Five ways designers fail to take advantage of language features and fail to respect limitations are worth mentioning:

[1] Ignore classes and express the design in a way that constrains implementers to use the C subset only.
[2] Ignore derived classes and virtual functions and use only the data abstraction subset.
[3] Ignore the static type checking and express the design in such a way that implementers are constrained to simulate dynamic type checking.
[4] Ignore programming and express systems in a way that aims to eliminate programmers.
[5] Ignore everything except class hierarchies.

These variants are typical for designers with

[1] a C, traditional CASE, or structured design background,
[2] an Ada83, Visual Basic, or data abstraction background,
[3] a Smalltalk or Lisp background,
[4] a nontechnical or very specialized background,
[5] a background with heavy emphasis on ''pure'' object-oriented programming,

respectively. In each case, one must wonder if the implementation language was well chosen, if the design method was well chosen, or if the designer had failed to adapt to the tool in hand.

There is nothing unusual or shameful in such a mismatch. It is simply a mismatch that delivers sub-optimal designs and imposes unnecessary burdens on programmers. It does the same to designers when the conceptual framework of the design method is noticeably poorer than C++'s conceptual framework. Therefore, we avoid such mismatches wherever possible.

The following discussion is phrased as answers to objections because that is the way it often occurs in real life.

24.2.1 Ignoring Classes [lang.ignore.class]

Consider design that ignores classes. The resulting C++ program will be roughly equivalent to the C program that would have resulted from the same design process – and this program would again be roughly equivalent to the COBOL program that would have resulted from the same design process. In essence, the design has been made ''programming language independent'' at the cost of forcing the programmer to code in the common subset of C and COBOL. This approach does have advantages. For example, the strict separation of data and code that results makes it easy to use traditional databases that are designed for such programs. Because a minimal programming language is used, it would appear that less skill – or at least different skills – would be required from programmers. For many applications – say, a traditional sequential database update program – this way of thinking is quite reasonable, and the traditional techniques developed over decades are adequate for the job.

However, suppose the application differs sufficiently from traditional sequential processing of records (or characters) or the complexity involved is higher – say, in an interactive CASE system.

The lack of language support for data abstraction implied by the decision to ignore classes will hurt. The inherent complexity will show up in the application somewhere, and if the system is implemented in an impoverished language, the code will not reflect the design directly. The program will have too many lines of source code, lack type checking, and will in general not be amenable to tools. This is the prescription for a maintenance nightmare.

A common band-aid for this problem is to build specific tools to support the notions of the design method. These tools then provide higher-level constructs and checking to compensate for deficiencies of the (deliberately impoverished) implementation language. Thus, the design method becomes a special-purpose and typically corporate-owned programming language. Such programming languages are in most contexts poor substitutes for a widely available, general-purpose programming language supported by suitable design tools.

The most common reason for ignoring classes in design is simple inertia. Traditional programming languages don't support the notion of a class, and traditional design techniques reflect this deficiency. The most common focus of design has been the decomposition of the problems into a set of procedures performing required actions. This notion, called procedural programming in Chapter 2, is in the context of design often called *functional decomposition*. A common question is, ''Can we use C++ together with a design method based on functional decomposition?'' You can, but you will most likely end up using C++ as simply a better C and will suffer the problems mentioned previously. This may be acceptable in a transition period, for already completed designs, and for subsystems in which classes do not appear to offer significant benefits (given the experience of the individuals involved at this time). For the longer term and in general, however, the policy against large-scale use of classes implied by functional decomposition is not compatible with effective use of C++ or any other language that has support for abstraction.

The procedure-oriented and object-oriented views of programming are fundamentally different and typically lead to radically different solutions to the same problem. This observation is as true for the design phase as it is for the implementation phase: you can focus the design on the actions taken or on the entities represented, but not simultaneously on both.

So why prefer ''object-oriented design'' over the traditional design methods based on functional decomposition? A first-order answer is that functional decomposition leads to insufficient data abstraction. From this, it follows that the resulting design is

- less resilient to change,
- less amenable to tools,
- less suited for parallel development, and
- less suited for concurrent execution.

The problem is that functional decomposition causes interesting data to become global because when a system is structured as a tree of functions, any data accessed by two functions must be global to both. This ensures that ''interesting'' data bubbles up toward the root of the tree as more and more functions require access to it (as ever in computing, trees grow from the root down). Exactly the same process can be seen in single-rooted class hierarchies, in which ''interesting'' data and functions tend to bubble up toward a root class (§24.4). Focussing on the specification of classes and the encapsulation of data addresses this problem by making the dependencies between different parts of a program explicit and tractable. More important, though, it reduces the number of dependencies in a system by improving locality of reference to data.

However, some problems are best solved by writing a set of procedures. The point of an ''object-oriented'' approach to design is not that there should never be any nonmember functions in a program or that no part of a system may be procedure-oriented. Rather, the key point is to decouple different parts of a program to better reflect the concepts of the application. Typically, that is best done when classes, not functions, are the primary focus on the design effort. The use of a procedural style should be a conscious decision and not simply a default. Both classes and procedures should be used appropriately relative to the application and not just as artifacts of an inflexible design method.

24.2.2 Avoiding Inheritance [lang.avoid.hier]

Consider design that avoids inheritance. The resulting programs simply fail to take advantage of a key C++ feature, while still reaping many benefits of C++ compared to C, Pascal, Fortran, COBOL, etc. Common reasons for doing this – apart from inertia – are claims that ''inheritance is an implementation detail,'' ''inheritance violates information hiding,'' and ''inheritance makes cooperation with other software harder.''

Considering inheritance merely an implementation detail ignores the way that class hierarchies can directly model key relationships between concepts in the application domain. Such relationships should be explicit in the design to allow designers to reason about them.

A strong case can be made for excluding inheritance from the parts of a C++ program that must interface directly with code written in other languages. This is, however, *not* a sufficient reason for avoiding the use of inheritance throughout a system; it is simply a reason for carefully specifying and encapsulating a program's interface to ''the outer world.'' Similarly, worries about compromising information hiding through the use of inheritance (§24.3.2.1) are a reason to be careful with the use of virtual functions and protected members (§15.3). They are not a reason for general avoidance.

In many cases, there is no real advantage to be gained from inheritance. However, in a large project a policy of ''no inheritance'' will result in a less comprehensible and less flexible system in which inheritance is ''faked'' using more traditional language and design constructs. Further, I suspect that despite such a policy, inheritance will eventually be used anyway because C++ programmers will find convincing arguments for inheritance-based designs in various parts of the system. Therefore, a ''no inheritance'' policy will ensure only that a coherent overall architecture will be missing and will restrict the use of class hierarchies to specific subsystems.

In other words, keep an open mind. Class hierarchies are not an essential part of every good program, but in many cases they can help in both the understanding of the application and the expression of a solution. The fact that inheritance can be misused and overused is a reason for caution; it is a not reason for prohibition.

24.2.3 Ignoring Static Type Checking [lang.type]

Consider design that ignores static type checking. Commonly stated reasons to ignore static type checking in the design phase are that ''types are an artifact of the programming language,'' that ''it is more natural to think about objects without bothering about types,'' and that ''static type checking forces us to think about implementation issues too early.'' This attitude is fine as far as it goes and harmless up to a point. It is reasonable to ignore details of type checking in the design stage,

and it is often safe to ignore type issues almost completely in the analysis stage and early design stages. However, classes and class hierarchies are very useful in the design. In particular, they allow us to be specific about concepts, allow us to be precise about their relationships, and help us reason about the concepts. As the design progresses, this precision takes the form of increasingly precise statements about classes and their interfaces.

It is important to realize that precisely-specified and strongly-typed interfaces are a fundamental design tool. C++ was designed with this in mind. A strongly-typed interface ensures (up to a point) that only compatible pieces of software can be compiled and linked together and thus allows these pieces of software to make relatively strong assumptions about each other. These assumptions are guaranteed by the type system. The effect of this is to minimize the use of run-time tests, thus promoting efficiency and causing significant reductions in the integration phase of multiperson projects. In fact, strong positive experience with integrating systems that provide strongly-typed interfaces is the reason integration isn't a major topic of this chapter.

Consider an analogy. In the physical world, we plug gadgets together all the time, and a seemingly infinite number of standards for plugs exists. The most obvious thing about these plugs is that they are specifically designed to make it impossible to plug two gadgets together unless the gadgets were designed to be plugged together, and then they can be connected only in the right way. You cannot plug an electric shaver into a high-power socket. Had you been able to, you would have ended up with a fried shaver or a fried shavee. Much ingenuity is expended on ensuring that incompatible pieces of hardware cannot be plugged together. The alternative to using many incompatible plugs is gadgets that protect themselves against undesirable behavior from gadgets plugged into their sockets. A surge protector is a good example of this. Because perfect compatibility cannot be guaranteed at the ''plug compatibility level,'' we occasionally need the more expensive protection of circuitry that dynamically adapts to and/or protects from a range of inputs.

The analogy is almost exact. Static type checking is equivalent to plug compatibility, and dynamic checking corresponds to protection/adaptation circuitry. If both checks fail – in either the physical world or the software world – serious damage can result. In large systems, both forms of checking are used. In the early stages of a design, it may be reasonable simply to say, ''These two gadgets should be plugged together.'' However, it soon becomes relevant exactly how they should be plugged together. What guarantees does the plug provide about behavior? What error conditions are possible? What are the first-order cost estimates?

The use of ''static typing'' is not limited to the physical world. The use of units (for example, meters, kilograms, and seconds) to prevent the mixing of incompatible entities is pervasive in physics and engineering.

In the description of the design steps in §23.4.3, type information enters the picture in Step 2 (presumably after being superficially considered in Step 1) and becomes a major issue in Step 4.

Statically-checked interfaces are the prime vehicle for ensuring cooperation between C++ software developed by different groups. The documentation of these interfaces (including the exact types involved) is the primary means of communication between separate groups of programmers. These interfaces are one of the most important outputs of the design process and a focus of communication between designers and programmers.

Ignoring type issues when considering interfaces leads to designs that obscure the structure of the program and postpone error detection until run time. For example, an interface can be specified in terms of self-identifying objects:

```
// Example assuming dynamic type checking instead of static checking:

Stack s; // Stack can hold pointers to objects of any type

void f()
{
    s.push(new Saab900);
    s.push(new Saab37B);

    s.pop()->takeoff();      // fine: a Saab 37B is a plane
    s.pop()->takeoff();      // run-time error: car cannot take off
}
```

This is a severe underspecification of the interface (of ***Stack*::*push*()**) that forces dynamic checking rather than static checking. The stack ***s*** is meant to hold ***Plane***s, but that was left implicit in the code, so it becomes the user's obligation to make sure the requirement is upheld.

A more precise specification – a template plus virtual functions rather than unconstrained dynamic type checking – moves error detection from run time to compile time:

```
Stack<Plane*> s; // Stack can hold pointers to Planes

void f()
{
    s.push(new Saab900);     // error: a Saab900 is not a Plane
    s.push(new Saab37B);

    s.pop()->takeoff();      // fine: a Saab 37B is a plane
    s.pop()->takeoff();
}
```

A similar point is made in §16.2.2. The difference in run time between dynamic checking and static checking can be significant. The overhead of dynamic checking is usually a factor in the range of 3 to 10.

One should not go to the other extreme, though. It is not possible to catch all errors by static checking. For example, even the most thoroughly statically checked program is vulnerable to hardware failures. See also §25.4.1 for an example where complete static checking would be infeasible. However, the ideal is to have the vast majority of interfaces be statically typed with application-level types; see §24.4.2.

Another problem is that a design can be perfectly reasonable in the abstract but can cause serious trouble because it fails to take into account limitations of a basic tool, in this case C++. For example, a function ***f*()** that needs to perform an operation ***turn_right*()** on an argument can do so only provided all of its arguments are of a common type:

```
class Plane {
    // ...
    void turn_right();
};
```

```
class Car {
    // ...
    void turn_right();
};

void f(X* p)    // what type should X be?
{
    p->turn_right();
    // ...
}
```

Some languages (such as Smalltalk and CLOS) allow two types to be used interchangeably if they have the same operations by relating every type through a common base and postponing name resolution until run time. However, C++ (intentionally) supports this notion through templates and compile-time resolution only. A non-template function can accept arguments of two types only if the two types can be implicitly converted to a common type. Thus, in the previous example ***X*** must be a common base of ***Plane*** and ***Car*** (e.g., a ***Vehicle*** class).

Typically, examples inspired by notions alien to C++ *can* be mapped into C++ by expressing the assumptions explicitly. For example, given ***Plane*** and ***Car*** (without a common base), we can still create a class hierarchy that allows us to pass an object containing a ***Car*** or a ***Plane*** to ***f(X*)*** (§25.4.1). However, doing this often requires an undesirable amount of mechanism and cleverness. Templates are often a useful tool for such concept mappings. A mismatch between design notions and C++ typically leads to ''unnatural-looking'' and inefficient code. Maintenance programmers tend to dislike the non-idiomatic code that arises from such mismatches.

A mismatch between the design technique and the implementation language can be compared to word-for-word translation between natural languages. For example, English with German grammar is as awkward as German with English grammar, and both can be close to incomprehensible to someone fluent in only one of those languages.

Classes in a program are the concrete representation of the concepts of the design. Consequently, obscuring the relationships between the classes obscures the fundamental concepts of the design.

24.2.4 Avoiding Programming [lang.prog]

Programming is costly and unpredictable compared to many other activities, and the resulting code is often less than 100% reliable. Programming is labor-intensive and – for a variety of reasons – most serious project delays manifest themselves by code not being ready to ship. So, why not eliminate programming as an activity altogether?

To many managers, getting rid of the arrogant, undisciplined, over-paid, technology-obsessed, improperly-dressed, etc. programmers† would appear to be a significant added benefit. To a programmer, this suggestion may sound absurd. However, important problem areas with realistic alternatives to traditional programming do exist. For specific areas, it is possible to generate code directly from a high-level specification. In other areas, code can be generated by manipulating shapes on a screen. For example, useful user interfaces can be constructed by direct manipulation

† Yes, I'm a programmer.

in a tiny fraction of the time it would take to construct the same interface by writing traditional code. Similarly, database layouts and the code for accessing data according to such layouts can be generated from specifications that are far simpler than the code needed to express those operations directly in C++ or in any other general-purpose programming language. State machines that are smaller, faster, and more correct than most programmers could produce can be generated from specifications or by a direct manipulation interface.

These techniques work well in specific areas where there is either a sound theoretical foundation (e.g., math, state machines, and relational databases) or where a general framework exists into which small application fragments can be embedded (e.g., graphical user interfaces, network simulations, and database schema). The obvious usefulness of these techniques in limited – and typically crucial – areas can tempt people to think that the elimination of traditional programming by these techniques is ''just around the corner.'' It is not. The reason is that expanding specification techniques outside areas with sound theoretical frameworks implies that the complexity of a general-purpose programming language would be needed in the specification language. This defeats the purpose of a clean and well-founded specification language.

It is sometimes forgotten that the framework that allows elimination of traditional programming in an area is a system or library that has been designed, programmed, and tested in the traditional way. In fact, one popular use of C++ and the techniques described in this book is to design and build such systems.

A compromise that provides a small fraction of the expressiveness of a general-purpose language is the worst of both worlds when applied outside a restricted application domain. Designers who stick to a high-level modeling point of view are annoyed by the added complexity and produce specifications from which horrendous code is produced. Programmers who apply ordinary programming techniques are frustrated by the lack of language support and generate better code only by excessive effort and by abandoning high-level models.

I see no signs that programming as an activity can be successfully eliminated outside areas that either have well-founded theoretical bases or in which the basic programming is provided by a framework. In either case, there is a dramatic drop in the effectiveness of the techniques as one leaves the original framework and attempts more general-purpose work. Pretending otherwise is tempting and dangerous. Conversely, ignoring the high-level specification techniques and the direct-manipulation techniques in domains in which they are well-founded and reasonably mature would be a folly.

Designing tools, libraries, and frameworks is one of the highest forms of design and programming. Constructing a useful mathematically-based model of an application area is one of the highest forms of analysis. Thus, providing a tool, language, framework, etc., that makes the result of such work available to thousands is a way for programmers and designers to escape the trap of becoming craftsmen of one-of-a-kind artifacts.

It is most important that a specification system or a foundation library be able to interface effectively with a general-purpose programming language. Otherwise, the framework provided is inherently limiting. This implies that specification systems and direct-manipulation systems that generate code at a suitable high level into an accepted general-purpose programming language have a great advantage. A proprietary language is a long-term advantage to its provider only. If the code generated is so low-level that general code added must be written without the benefits of abstraction, then reliability, maintainability, and economy are lost. In essence, a generation system should

be designed to combine the strengths of higher-level specifications and higher-level programming languages. To exclude one or the other is to sacrifice the interests of system builders to the interests of tool providers. Successful large systems are multilevel and modular and evolve over time. Consequently, successful efforts to produce such systems involve a variety of languages, libraries, tools, and techniques.

24.2.5 Using Class Hierarchies Exclusively [lang.pure]

When we find that something new actually works, we often go a bit overboard and apply it indiscriminately. In other words, a great solution to some problems often appears to be *the* solution to almost all problems. Class hierarchies and operations that are polymorphic on their (one) object provide a great solution to many problems. However, not every concept is best represented as a part of a hierarchy and not every software component is best represented as a class hierarchy.

Why not? A class hierarchy expresses relationships between its classes and a class represents a concept. Now what is the common relationship between a smile, the driver for my CD-ROM reader, a recording of Richard Strauss' Don Juan, a line of text, a satellite, my medical records, and a real-time clock? Placing them all in a single hierarchy when their only shared property is that they are programming artifacts (they are all ''objects'') is of little fundamental value and can cause confusion (§15.4.5). Forcing everything into a single hierarchy can introduce artificial similarities and obscure real ones. A hierarchy should be used only if analysis reveals conceptual commonality or if design and programming discover useful commonality in the structures used to implement the concepts. In the latter case, we have to be very careful to distinguish genuine commonality (to be reflected as subtyping by public inheritance) and useful implementation simplifications (to be reflected as private inheritance; §24.3.2.1).

This line of thinking leads to a program that has several unrelated or weakly-related class hierarchies, each representing a set of closely related concepts. It also leads to the notion of a concrete class (§25.2) that is not part of a hierarchy because placing such a class in a hierarchy would compromise its performance and its independence of the rest of the system.

To be effective, most critical operations on a class that is part of a class hierarchy must be virtual functions. Furthermore, much of that class' data must be protected rather than private. This makes it vulnerable to modification from further derived classes and can seriously complicate testing. Where stricter encapsulation makes sense from a design point of view, non-virtual functions and private data should be used (§24.3.2.1).

Having one argument of an operation (the one designating ''the object'') special can lead to contorted designs. When several arguments are best treated equally, an operation is best represented as a nonmember function. This does not imply that such functions should be global. In fact, almost all such free-standing functions should be members of a namespace (§24.4).

24.3 Classes [lang.class]

The most fundamental notion of object-oriented design and programming is that the program is a model of some aspects of reality. The classes in the program represent the fundamental concepts of the application and, in particular, the fundamental concepts of the ''reality'' being modeled. Real-world objects and artifacts of the implementation are represented by objects of these classes.

The analysis of relationships between classes and within parts of a class is central to the design of a system:

§24.3.2 Inheritance relationships
§24.3.3 Containment relationships
§24.3.5 Use relationships
§24.2.4 Programmed-in relationships
§24.3.7 Relationships within a class

Because a C++ class is a type, classes and the relationships between classes receive significant support from compilers and are generally amenable to static analysis.

To be relevant in a design, a class doesn't just have to represent a useful concept; it must also provide a suitable interface. Basically, the ideal class has a minimal and well-defined dependence on the rest of the world and presents an interface that exposes the minimal amount of information necessary to the rest of the world (§24.4.2).

24.3.1 What Do Classes Represent? [lang.what]

There are essentially two kinds of classes in a system:

[1] Classes that directly reflect the concepts in the application domain; that is, concepts that are used by end-users to describe their problems and solutions
[2] Classes that are artifacts of the implementation; that is, concepts that are used by the designers and programmers to describe their implementation techniques.

Some of the classes that are artifacts of the implementation may also represent real-world entities. For example, the hardware and software resources of a system provide good candidates for classes in an application. This reflects the fact that a system can be viewed from several viewpoints. This implies that one person's implementation detail is another person's application. A well-designed system will contain classes supporting logically separate views of the system. For example:

[1] Classes representing user-level concepts (e.g., cars and trucks)
[2] Classes representing generalizations of the user-level concepts (e.g. vehicles)
[3] Classes representing hardware resources (e.g., a memory management class)
[4] Classes representing system resources (e.g., output streams)
[5] Classes used to implement other classes (e.g., lists, queues, locks)
[6] Built-in data types and control structures.

In larger systems, keeping logically separate types of classes separate and maintaining separation between several levels of abstraction becomes a challenge. A simple example can be considered to have three levels of abstraction:

[1+2] Provide an application level view of the system
[3+4] Represent the machine on which the model runs
[5+6] Represent a low-level (programming language) view of the implementation.

The larger the system, the more levels of abstraction are typically needed for the description of the system and the more difficult it becomes to define and maintain the levels. Note that such levels of abstraction have direct counterparts in nature and in other types of human constructions. For example, a house can be considered as consisting of

[1] atoms;
[2] molecules;

[3] lumber and bricks;
[4] floors, walls, and ceilings; and
[5] rooms.

As long as these levels of abstraction are kept separate, you can maintain a coherent view of the house. However, if you mix them, absurdities arise. For example, the statement, ''My house consists of several thousand pounds of carbon, some complex polymers, about 5,000 bricks, two bathrooms, and 13 ceilings,'' is silly. Given the abstract nature of software, the equivalent statement about a complex system is not always recognized for what it is.

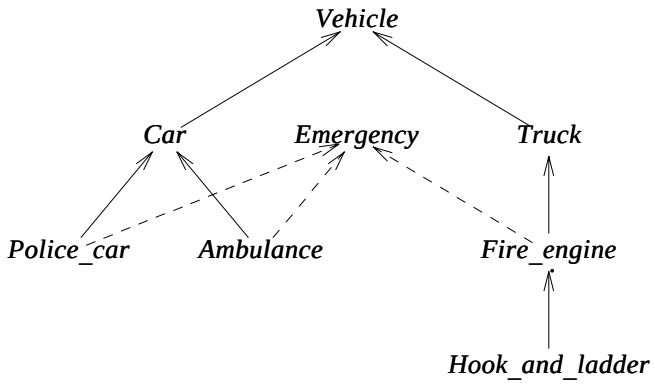
The translation of a concept in the application area into a class in a design is not a simple mechanical operation. It often requires significant insights. Note that the concepts in an application area are themselves abstractions. For example, ''taxpayers,'' ''monks,'' and ''employees'' don't really exist in nature; such concepts are themselves labels put on individuals to classify them relative to some system. The real or even the imagined world (literature, especially science fiction) is sometimes simply a source of ideas for concepts that mutate radically in the transition into classes. For example, the screen of my PC doesn't really resemble my desktop despite its being designed to support the desktop metaphor†, and the windows on my screen bear only the slightest relation to the contraptions that let drafts into my office. The point about modeling reality is not to slavishly follow what we see but rather to use it as a starting point for design, a source of inspiration, and an anchor to hold on to when the intangible nature of software threatens to overcome our ability to understand our programs.

A word of caution: beginners often find it hard to ''find the classes,'' but that problem is usually soon overcome without long-term ill effects. Next, however, often follows a phase in which classes – and their inheritance relationships – seem to multiply uncontrollably. This can cause long-term problems with the complexity, comprehensibility, and efficiency of the resulting program. Not every minute detail needs to be represented by a distinct class, and not every relationship between classes needs to be represented as an inheritance relationship. Try to remember that the aim of a design is to model a system at an *appropriate* level of detail and at *appropriate* levels of abstraction. Finding a balance between simplicity and generality is not easy.

24.3.2 Class Hierarchies [lang.hier]

Consider simulating the traffic flow of a city to determine the likely times needed for emergency vehicles to reach their destinations. Clearly, we need to represent cars, trucks, ambulances, fire engines of various sorts, police cars, busses, etc. Inheritance comes into play because a real-world concept does not exist in isolation; it exists with numerous relationships to other concepts. Without understanding these relationships, we cannot understand the concepts. Consequently, a model that does not represent such relationships does not adequately represent our concepts. That is, in our programs we need classes to represent concepts, but that is not enough. We also need ways of representing relationships between classes. Inheritance is one powerful way of representing hierarchical relationships directly. In our example, we would probably consider emergency vehicles special and want also to distinguish between car-like and truck-like vehicles. This would yield a class hierarchy along these lines:

† I wouldn't be able to tolerate such a mess on my screen, anyway.

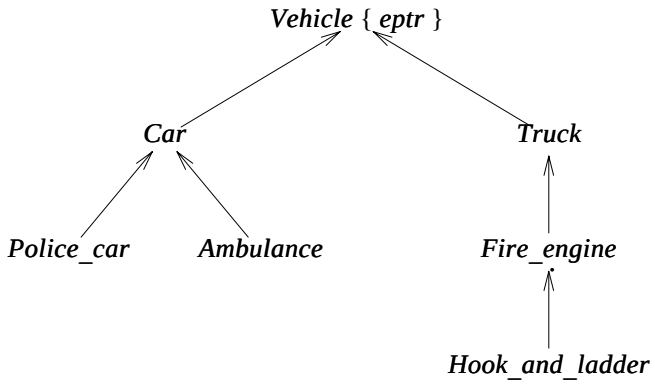


Here, ***Emergency*** represents the aspects of an emergency vehicle that are relevant to the simulation: it can violate some traffic rules, has priority in intersections when on an emergency call, it is under control of a dispatcher, etc.

Here is the C++ version:

```
class Vehicle { /* ... */ };
class Emergency { /* ... */ };
class Car : public Vehicle { /* ... */ };
class Truck : public Vehicle { /* ... */ };
class Police_car : public Car , protected Emergency { /* ... */ };
class Ambulance : public Car , protected Emergency { /* ... */ };
class Fire_engine : public Truck , protected Emergency { /* ... */ };
class Hook_and_ladder : public Fire_engine { /* ... */ };
```

Inheritance is the highest level relationship that can be represented directly in C++ and the one that figures largest in the early stages of a design. Often there is a choice between using inheritance to represent a relationship and using membership. Consider an alternative notion of what it means to be an emergency vehicle: a vehicle is an emergency vehicle if it displays a flashing light. This would allow a simplification of the class hierarchy by replacing the ***Emergency*** class by a member in class ***Vehicle*****:**



Class *Emergency* is now simply used as a member in classes that might need to act as emergency vehicles:

```
class Emergency { /* ... */ };
class Vehicle { protected: Emergency* eptr; /* ... */ };  // better: provide proper interface to eptr
class Car : public Vehicle { /* ... */ };
class Truck : public Vehicle { /* ... */ };
class Police_car : public Car { /* ... */ };
class Ambulance : public Car { /* ... */ };
class Fire_engine : public Truck { /* ... */ };
class Hook_and_ladder : public Fire_engine { /* ... */ };
```

Here, a vehicle is an emergency vehicle if *Vehicle::eptr* is nonzero. The ''plain'' cars and trucks are initialized with *Vehicle::eptr* zero; the others are initialized with *Vehicle::eptr* nonzero. For example:

```
Car::Car()      // Car constructor
{
    eptr = 0;
}

Police_car::Police_car()  // Police_car constructor
{
    eptr = new Emergency;
}
```

Defining things this way enables a simple conversion of an emergency vehicle to an ordinary vehicle and vice versa:

```
void f(Vehicle* p)
{
    delete p->eptr;
    p->eptr = 0;                    // no longer an emergency vehicle

    // ...

    p->eptr = new Emergency;        // an emergency vehicle again
}
```

So, which variant of the class hierarchy is best? The general answer is, ''The program that most directly models the aspects of the real world that we are interested in is the best.'' That is, in choosing between models we should aim for greater realism under the inevitable constraints of efficiency and simplicity. In this case, the easy conversion between ordinary vehicles and emergency vehicles seems unrealistic to me. Fire engines and ambulances are purpose-built vehicles manned by trained personnel and operated using dispatch procedures requiring specialized communication equipment. This view indicates that being an emergency vehicle should be a fundamental concept and represented directly in the program to improve type checking and other uses of tools. Had we been modeling a place where the roles of vehicles were less firmly defined – say, an area where private vehicles were routinely used to carry emergency personnel to accident sites and where communication was primarily based on portable radios – the other way of modeling the system might have been more appropriate.

For people who consider traffic simulations esoteric, it might be worth pointing out that such tradeoffs between inheritance and membership almost invariably occur in a design. The scrollbar example in §24.3.3 is an equivalent example.

24.3.2.1 Dependencies within a Class Hierarchy [lang.internal]

Naturally, a derived class depends on its base classes. It is less often appreciated that the opposite can also be true†. If a class has a virtual function, the class depends on derived classes to implement part of its functionality whenever a derived class overrides that function. If a member of a base class itself calls one of the class' virtual functions, then the base class depends on its derived classes for its own implementation. Similarly, if a class uses a protected member, then it is again dependent on its derived classes for its own implementation. Consider:

```
class B {
    // ...
protected:
    int a;
public:
    virtual int f();
    int g() { int x = f(); return x-a; }
};
```

What does `g()` do? The answer critically depends on the definition of `f()` in some derived class. Here is a version that will ensure that `g()` returns `1`:

```
class D1 : public B {
    int f() { return a+1; }
};
```

and a version that makes `g()` write "`Hello, world!`" and return `0`:

```
class D2 : public B {
    int f() { cout<<"Hello, world!\n"; return a; }
};
```

This example illustrates one of the most important points about virtual functions. Why is it silly? Why wouldn't a programmer ever write something like that? The answer is that a virtual function is part of an interface to a base class, and that class can supposedly be used without knowledge of the classes derived from it. Consequently, it must be possible to describe the expected behavior of an object of the base class in such a way that programs can be written without knowledge of the derived classes. Every class that overrides the virtual function must implement a variant of that behavior. For example, the virtual function *rotate*() of a *Shape* class rotates a shape. The *rotate*() functions for derived classes such as *Circle* and *Triangle* must rotate objects of their respective type; otherwise, a fundamental assumption about class *Shape* is violated. No such assumption about behavior is made for class *B* or its derived classes *D1* and *D2*; thus, the example is nonsensical. Even the names *B*, *D1*, *D2*, *f*, and *g* were chosen to obscure any possible meanings.

† This observation has been summarized as: "Insanity is hereditary. You get it from your children."

The specification of the expected behavior of virtual functions is a *major* focus of class design. Choosing good names for classes and functions is important – and not always easy.

Is a dependency on unknown (possibly yet unwritten) derived classes good or bad? Naturally, that depends on the intent of the programmer. If the intent is to isolate a class from all external influences so that it can be proven to behave in a specific way, then protected members and virtual functions are best avoided. If, however, the intent is to provide a framework into which a later programmer (such as the same programmer a few weeks later) can add code, then virtual functions are often an elegant mechanism for achieving this; and protected member functions have proven convenient for supporting such use. This technique is used in the stream I/O library (§21.6) and was illustrated by the final version of the ***Ival_box*** hierarchy (§12.4.2).

If a ***virtual*** function is meant to be used only indirectly by a derived class, it can be left ***private***. For example, consider a simple buffer template:

```
template<class T> class Buffer {
public:
    void put(T);    // call overflow(T) if buffer is full
    T get();        // call underflow() if buffer is empty
    // ...
private:
    virtual int overflow(T);
    virtual int underflow();
    // ...
};
```

The ***put***() and ***get***() functions call ***virtual*** functions ***overflow***() and ***underflow***(), respectively. A user can now implement a variety of buffer types to suit a variety of needs by overriding ***overflow***() and ***underflow***():

```
template<class T> class Circular_buffer : public Buffer<T> {
    int overflow(T);    // wrap around if full
    int underflow();
    // ...
};

template<class T> class Expanding_buffer : public Buffer<T> {
    int overflow(T);    // increase buffer size if full
    int underflow();
    // ...
};
```

Only if a derived class needed to call ***overflow***() and ***underflow***() directly would these functions need to be ***protected*** rather than ***private***.

24.3.3 Containment Relationships [lang.contain]

Where containment is used, there are two major alternatives for representing an object of a class ***X***:

[1] Declare a member of type ***X***.

[2] Declare a member of type ***X****** or type ***X***&.

If the value of the pointer is never changed, these alternatives are equivalent, except for efficiency issues and the way you write constructors and destructors:

```
class X {
public:
    X(int);
    // ...
};

class C {
    X a;
    X* p;
    X& r;
public:
    C(int i, int j, int k) : a(i), p(new X(j)), r(*new X(k)) { }
    ~C() { delete p; delete &r; }
};
```

In such cases, membership of the object itself, as in the case of `C::a`, is usually preferable because it is the most efficient in time, space, and keystrokes. It is also less error-prone because the connection between the contained object and the containing object is covered by the rules of construction and destruction (§10.4.1, §12.2.2, §14.4.1). However, see also §24.4.2 and §25.7.

The pointer solution should be used when there is a need to change the pointer to the ''contained'' object during the life of the ''containing'' object. For example:

```
class C2 {
    X* p;
public:
    C2(int i) : p(new X(i)) { }
    ~C2() { delete p; }

    X* change(X* q)
    {
        X* t = p;
        p = q;
        return t;
    }
};
```

Another reason for using a pointer member is to allow the ''contained'' member to be supplied as an argument:

```
class C3 {
    X* p;
public:
    C3(X* q) : p(q) { }
    // ...
};
```

By having objects contain pointers to other objects, we create what are often called *object hierarchies*. This is an alternative and complementary technique to using class hierarchies. As shown in the emergency vehicle example in §24.3.2, it is often a tricky design issue to choose

between representing a property of a class as a base class or representing it as a member. A need to override is an indication that the former is the better choice. Conversely, a need to be able to allow the property to be represented by a variety of types is an indication that the latter is the better choice. For example:

```
class XX : public X { /* ... */ };

class XXX : public X { /* ... */ };

void f()
{
    C3* p1 = new C3(new X);      // C3 ''contains'' an X
    C3* p2 = new C3(new XX);     // C3 ''contains'' an XX
    C3* p3 = new C3(new XXX);    // C3 ''contains'' an XXX
    // ...
}
```

This could not be modeled by a derivation of ***C3*** from ***X*** or by ***C3*** having a member of type ***X***, because the exact type of a member needs to be used. This is important for classes with virtual functions, such as a shape class (§2.6.2) or an abstract set class (§25.3).

References can be used to simplify classes based on pointer membership when only one object is referred to during the life of the containing object. For example:

```
class C4 {
    X& r;
public:
    C4(X& q) : r(q) { }
    // ...
};
```

Pointer and reference members are also needed when an object needs to be shared:

```
X* p = new XX;
C4 obj1(*p);
C4 obj2(*p);   // obj1 and obj2 now share the new XX
```

Naturally, management of shared objects requires extra care – especially in concurrent systems.

24.3.4 Containment and Inheritance [lang.cont.hier]

Given the importance of inheritance relationships, it is not surprising that they are frequently overused and misunderstood. When a class ***D*** is publicly derived from another class ***B***, it is often said that a ***D*** *is* a ***B***:

```
class B { /* ... */ };
class D : public B { /* ... */ };      // D is a kind of B
```

Alternatively, this is expressed by saying that inheritance is an *is-a* relationship or – somewhat more precisely – that a ***D*** *is a kind of* ***B***. In contrast, a class ***D*** that has a member of another class ***B*** is often said to *have* a ***B*** or *contain* a ***B***. For example:

```
class D { // a D contains a B
public:
        B b;
        // ...
};
```

Alternatively, this is expressed by saying that membership is a *has-a* relationship.

For given classes ***B*** and ***D***, how do we choose between inheritance and membership? Consider an ***Airplane*** and an ***Engine***. Novices often wonder if it might be a good idea to derive class ***Airplane*** from ***Engine***. This is a bad idea, though, because an ***Airplane*** *is* not an ***Engine***; it *has* an ***Engine***. One way of seeing this is to consider if an ***Airplane*** might have two or more engines. Because that seems feasible (even if we are considering a program in which all of our ***Airplane***s will be single-engine ones), we should use membership rather than inheritance. The question ''can it have two?'' is useful in many cases when there is doubt. As usual, it is the intangible nature of software that makes this discussion relevant. Had all classes been as easy to visualize as ***Airplane*** and ***Engine***, trivial mistakes like deriving an ***Airplane*** from an ***Engine*** would be easily avoided. Such mistakes are, however, quite frequent – particularly among people who consider derivation as simply another mechanism for combining programming-language-level constructs. Despite the conveniences and shorthand notation that derivation provides, it should be used almost exclusively to express relationships that are well defined in a design. Consider:

```
class B {
public:
        virtual void f();
        void g();
};

class D1 {              // a D1 contains a B
public:
        B b;
        void f();       // does not override b.f()
};

void h1(D1* pd)
{
        B* pb = pd;     // error: no D1* to B* conversion
        pb = &pd->b;
        pb->g();        // calls B::g()
        pd->g();        // error: D1 doesn't have a member g()
        pd->b.g();
        pb->f();        // calls B::f (not overridden by D1::f())
        pd->f();        // calls D1::f()
}
```

Note that there is no implicit conversion from a class to one of its members and that a class containing a member of another class does not override the virtual functions of that member. This contrasts with the public derivation case:

```
class D2 : public B {      // a D2 is a B
public:
    void f();        // overrides B::f()
};

void h2(D2* pd)
{
    B* pb = pd;      // ok: implicit D2* to B* conversion
    pb->g();         // calls B::g()
    pd->g();         // calls B::g()
    pb->f();         // virtual call: invokes D2::f()
    pd->f();         // invokes D2::f()
}
```

The notational convenience provided by the ***D2*** example compared to the ***D1*** example is a factor that can lead to overuse. It should be remembered, though, that there is a cost of increased dependency between ***B*** and ***D2*** to be paid for that notational convenience (see §24.3.2.1). In particular, it is easy to forget the implicit conversion from ***D2*** to ***B***. Unless such conversions are an acceptable part of the semantics of your classes, *public* derivation is to be avoided. When a class is used to represent a concept and derivation is used to represent an *is-a* relationship, such conversions are most often exactly what is desired.

There are cases in which you would like inheritance but cannot afford to have the conversion happen. Consider writing a class ***Cfield*** (controlled field) that – in addition to whatever else it does – provides run-time access control for another class ***Field***. At first glance, defining ***Cfield*** by deriving it from ***Field*** seems just right:

```
class Cfield : public Field { /* ... */ };
```

This expresses the notion that a ***Cfield*** really is a kind of ***Field***, allows notational convenience when writing a ***Cfield*** function that uses a member of the ***Field*** part of the ***Cfield***, and – most importantly – allows a ***Cfield*** to override ***Field*** virtual functions. The snag is that the ***Cfield****** to ***Field****** conversion implied in the declaration of ***Cfield*** defeats all attempts to control access to the ***Field***:

```
void g(Cfield* p)
{
    *p = "asdf";     // access to Field controlled by Cfield's assignment operator:
                     // p->Cfield::operator=("asdf")

    Field* q = p;    // implicit Cfield* to Field* conversion
    *q = "asdf";     // OOPS! no control
}
```

A solution would be to define ***Cfield*** to have a ***Field*** as a member, but doing that precludes ***Cfield*** from overriding ***Field*** virtual functions. A better solution would be to use *private* derivation:

```
class Cfield : private Field { /* ... */ };
```

From a design perspective, private derivation is equivalent to containment, except for the (occasionally essential) issue of overriding. An important use of this is the technique of deriving a class

publicly from an abstract base class that defines an interface and using private or protected derivation from a concrete class to provide an implementation (§2.5.4, §12.3, §25.3). Because the inheritance implied in *private* and *protected* derivation is an implementation detail that is not reflected in the type of the derived class, it is sometimes called *implementation inheritance* and contrasted to *public* derivation, whereby the interface of the base class is inherited and the implicit conversion to the base type is allowed. The latter is sometimes referred to as *subtyping*, or *interface inheritance*.

Another way of stating this is to point out that an object of a derived class should be usable wherever an object of its public base class is. This is sometimes called ''the Liskov Substitution Principle'' (§23.6[Liskov,1987]). The public/protected/private distinction supports this directly for polymorphic types manipulated through pointers and references.

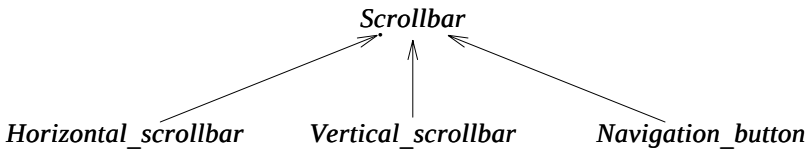
24.3.4.1 Member/Hierarchy Tradeoffs [lang.mem]

To further examine the design choices involving containment and inheritance, consider how to represent a scrollbar in an interactive graphics system and how to attach a scrollbar to a window. We need two kinds of scrollbars: horizontal and vertical. We can represent this either by two types – ***Horizontal_scrollbar*** and ***Vertical_scrollbar*** – or by a single ***Scrollbar*** type that takes an argument that says whether its layout is horizontal or vertical. The former choice implies the need for a third type, the plain ***Scrollbar***, as the base class of the two specific scollbar types. The latter choice implies the need for an extra argument to the scrollbar type and the need to choose values to represent the two kinds of scrollbars. For example:

```
enum Orientation { horizontal, vertical };
```

Once a choice is made, it determines the kind of change needed to extend the system. In the scrollbar example, we might want to introduce a third type of scrollbar. We may originally have thought that there could be only two kinds of scrollbars (''after all, a window has only two dimensions''). However, in this case – as in most – there are possible extensions that surface as redesign issues. For example, one might like to use a ''navigation button'' instead of two scrollbars. Such a button would cause scrolling in different directions depending on where a user pressed it. Pressing the middle of the top would cause ''scrolling up,'' pressing the middle left would cause ''scrolling left,'' while pressing the top-left corner would cause ''scrolling up and left.'' Such buttons are not uncommon. They can be seen as a refinement of the notion of a scrollbar that is particularly suited to applications in which the information scrolled over isn't plain text but rather more general sorts of pictures.

Adding a navigation button to a program with a three-scrollbar class hierarchy involves adding a new class, but it requires no changes to the old scrollbar code:



This is the nice aspect of the ''hierarchical'' solution.

Passing the orientation of the scrollbar as an argument implies the presence of type fields in the scrollbar objects and the use of switch statements in the code of the scrollbar member functions.

That is, we are facing a tradeoff between expressing this aspect of the structure of the system in terms of declarations or in terms of code. The former increases the degree of static checking and the amount of information on which tools have to work. The latter postpones decisions to run time and allows changes to be made by modifying individual functions without affecting the overall structure of the system as seen by the type checker and other tools. In most situations, I recommend using a class hierarchy to directly model hierarchical relationships of the concepts.

The single scrollbar type solution makes it easy to store and pass information specifying a kind of scrollbar:

```
void helper(Orientation oo)
{
    // ...
    p = new Scrollbar(oo);
    // ...
}

void me()
{
    helper(horizontal);
    // ...
}
```

This representation would also make it easy to re-orient a scrollbar at run time. This is unlikely to be of major importance in the case of scrollbars, but it can be important for equivalent examples. The point here is that there are always tradeoffs, and the tradeoffs are often nontrivial.

24.3.4.2 Containment/Hierarchy Tradeoffs [lang.tradeoff]

Now consider how to attach a scrollbar to a window. If we consider a ***Window_with_scrollbar*** as something that is both a ***Window*** and a ***Scrollbar***, we get something like:

```
class Window_with_scrollbar : public Window, public Scrollbar {
    // ...
};
```

This allows any ***Window_with_scrollbar*** to act like a ***Scrollbar*** and like a ***Window***, but it constrains us to using the single scrollbar-type solution.

On the other hand, if we consider a ***Window_with_scrollbar*** as a ***Window*** that has a ***Scrollbar***, we get something like:

```
class Window_with_scrollbar : public Window {
    // ...
    Scrollbar* sb;
public:
    Window_with_scrollbar(Scrollbar* p, /* ... */) : Window(/* ... */), sb(p) { /* ... */ }
    // ...
};
```

This allows us to use the scrollbar-hierarchy solution. Passing the scrollbar as an argument allows the window to be oblivious to the exact type of its scrollbar. We could even pass a ***Scrollbar***

around the way we passed an ***Orientation*** (§24.3.4.1). If we need to have ***Window_with_scrollbar*** act as a scrollbar, we can add a conversion operator:

```
Window_with_scrollbar::operator Scrollbar&()
{
    return *sb;
}
```

My preference is to have a window contain a scrollbar. I find it easier to think of a window *having* a scrollbar than of a window *being* a scrollbar in addition to being a window. In fact, my favorite design strategy involves a scrollbar being a special kind of window, which is then contained in a window that needs scrollbar services. This strategy forces the decision in favor of the containment solution. An alternative argument for the containment solution comes from the ''can it have two?'' rule of thumb (§24.3.4). Because there is no logical reason why a window shouldn't have two scrollbars (in fact, many windows do have both a horizontal and a vertical scrollbar), ***Window_with_scrollbar*** ought not be derived from ***Scrollbar***.

Note that it is not possible to derive from an unknown class. The exact type of a base class must be known at compile time (§12.2). On the other hand, if an attribute of a class is passed as an argument to its constructor, then somewhere in the class there must be a member that represents it. However, if that member is a pointer or a reference we can pass an object of a class derived from the class specified for the member. For example, The ***Scrollbar**** member ***sb*** in the previous example can point to a ***Scrollbar*** of a type, such as ***Navigation_button***, that is unknown to users of the ***Scrollbar****.

24.3.5 Use Relationships [lang.use]

Knowledge of what other classes are used by a class and in which ways is often critical in order to express and understand a design. Such dependencies are supported only implicitly by C++. A class can use only names that have been declared (somewhere), but a list of names used is not provided in the C++ source. Tools (or in the absence of suitable tools, careful reading) are necessary for extracting such information. The ways a class ***X*** can use another class ***Y*** can be classified in several ways. Here is one way:

- ***X*** uses the name ***Y***.
- ***X*** uses ***Y***.
  - ***X*** calls a ***Y*** member function.
  - ***X*** reads a member of ***Y***.
  - ***X*** writes a member of ***Y***.
- ***X*** creates a ***Y***.
  - ***X*** allocates an ***auto*** or ***static*** variable of ***Y***.
  - ***X*** creates a ***Y*** using ***new***.
- ***X*** takes the size of a ***Y***.

Taking the size of an object is classified separately because doing so requires knowledge of the class declaration, but doesn't depend on the constructors. Naming ***Y*** is also classified as a separate dependency because just doing that – for example, in declaring a ***Y**** or mentioning ***Y*** in the declaration of an external function – doesn't require access to the declaration of ***Y*** at all (§5.7):

```
class Y;   // Y is the name of a class
Y* p;
extern Y f(const Y&);
```

It is often important to distinguish between the dependencies of a class' interface (the class declaration) and the dependencies of the class implementation (the class member definitions). In a well-designed system, the latter typically have many more dependencies, and those are far less interesting to a user than are the dependencies of the class declaration (§24.4.2). Typically, a design aims at minimizing the dependencies of an interface because they become dependencies of the class' users (§8.2.4.1, §9.3.2, §12.4.1.1, §24.4) .

C++ doesn't require the implementer of a class to specify in detail what other classes are used and how. One reason for this is that most significant classes depend on so many other classes, that an abbreviation of the list of those classes, such as an ***#include*** directive, would be necessary for readability. Another is that the classification and granularity of such dependencies doesn't appear to be a programming language issue. Rather, exactly how *uses* dependencies are viewed depends on the purpose of the designer, programmer, or tool. Finally, which dependencies are interesting may also depend on details of the language implementation.

24.3.6 Programmed-In Relationships [lang.prog]

A programming language cannot – and should not – directly support every concept from every design method. Similarly, a design language should not support every feature of every programming language. A design language should be richer and less concerned with details than a language suitable for systems programming must be. Conversely, a programming language must be able to support a variety of design philosophies, or it will fail for lack of adaptability.

When a programming language does not provide facilities for representing a concept from the design directly, a conventional mapping between the design construct and the programming language constructs should be used. For example, a design method may have a notion of delegation. That is, the design can specify that every operation not defined for a class ***A*** should be serviced by an object of a class ***B*** pointed to by a pointer ***p***. C++ cannot express this directly. However, the expression of that idea in C++ is so stylized that one could easily imagine a program generating the code. Consider:

```
class B {
      // ...
      void f();
      void g();
      void h();
};

class A {
      B* p;
      // ...
      void f();
      void ff();
};
```

A specification that ***A*** delegated to ***B*** through ***A::p*** would result in code like this:

```
class A {
    B* p;       // delegation through p
    // ...
    void f();
    void ff();
    void g() { p->g(); }      // delegate g()
    void h() { p->h(); }      // delegate h()
};
```

It is fairly obvious to a programmer what is going on here, but simulating a design concept in code is clearly inferior to a one-to-one correspondence. Such ''programmed-in'' relationships are not as well ''understood'' by the programming language and are therefore less amenable to manipulation by tools. For example, standard tools would not recognize the ''delegation'' from ***A*** to ***B*** through ***A::p*** as different from any other use of a ***B****.

A one-to-one mapping between the design concepts and the programming language concepts should be used wherever possible. A one-to-one mapping ensures simplicity and guarantees that the design really is reflected in the program so that programmers and tools can take advantage of it.

Conversion operators provide a language mechanism for expressing a class of programmed-in relationships. That is, a conversion operator ***X::operator Y()*** specifies that wherever a ***Y*** is acceptable, an ***X*** can be used (§11.4.1). A constructor ***Y::Y(X)*** expresses the same relationship. Note that a conversion operator (and a constructor) produces a new object rather than changing the type of an existing object. Declaring a conversion function to ***Y*** is simply a way of requesting *implicit* application of a function that returns a ***Y***. Because the implicit application of conversions defined by constructors and conversion operators can be treacherous, it is sometimes useful to analyze them separately in a design.

It is important to ensure that the conversion graphs for a program do not contain cycles. If they do, the resulting ambiguity errors will render the types involved in the cycles unusable in combination. For example:

```
class Rational;

class Big_int {
public:
    friend Big_int operator+(Big_int,Big_int);
    operator Rational();
    // ...
};

class Rational {
public:
    friend Rational operator+(Rational,Rational);
    operator Big_int();
    // ...
};
```

The ***Rational*** and ***Big_int*** types will not interact as smoothly as one might have hoped:

```
void f(Rational r, Big_int i)
{
    g(r+i);              // error, ambiguous: operator+(r,Rational(i)) or operator+(Big_int(r),i) ?
    g(r+Rational(i));    // one explicit resolution
    g(Big_int(r)+i);     // another explicit resolution
}
```

One can avoid such ''mutual'' conversions by making at least some of them explicit. For example, the ***Big_int*** to ***Rational*** conversion might have been defined as ***make_Rational***`()` instead of as a conversion operator, and the addition would have been resolved to `g(`***Big_int***`(r),i)`. Where ''mutual'' conversion operators cannot be avoided, one must resolve the resulting clashes either by explicit conversions as shown or by defining many separate versions of binary operators, such as `+`.

24.3.7 Relationships within a Class [lang.within]

A class can conceal just about any implementation detail and just about any amount of dirt – and sometimes it has to. However, the objects of most classes do themselves have a regular structure and are manipulated in ways that are fairly easy to describe. An object of a class is a collection of other sub-objects (often called members), and many of these are pointers and references to other objects. Thus, an object can be seen as the root of a tree of objects and the objects involved can be seen as constituting an ''object hierarchy'' that is complementary to the class hierarchy, as described in §24.3.2.1. For example, consider a very simple ***String***:

```
class String {
    int sz;
    char* p;
public:
    String(const char* q);
    ~String();
    // ...
};
```

A ***String*** object can be represented graphically like this:

24.3.7.1 Invariants [lang.invariant]

The values of the members and the objects referred to by members are collectively called the *state* of the object (or simply, its *value*). A major concern of a class design is to get an object into a well-defined state (initialization/construction), to maintain a well-defined state as operations are performed, and finally to destroy the object gracefully. The property that makes the state of an object well-defined is called its *invariant*.

Thus, the purpose of initialization is to put an object into a state for which the invariant holds. Typically, this is done by a constructor. Each operation on a class can assume it will find the invariant true on entry and must leave the invariant true on exit. The destructor finally invalidates the invariant by destroying the object. For example, the constructor *String::String(const char*)* ensures that *p* points to an array of at least *sz+1* elements, where *sz* has a reasonable value and *p[sz]==0*. Every string operation must leave that assertion true.

Much of the skill in class design involves making a class simple enough to make it possible to implement it so that it has a useful invariant that can be expressed simply. It is easy enough to state that every class needs an invariant. The hard part is to come up with a useful invariant that is easy to comprehend and that doesn't impose unacceptable constraints on the implementer or on the efficiency of the operations. Note that ''invariant'' here is used to denote a piece of code that can potentially be run to check the state of an object. A stricter and more mathematical notion is clearly possible and, in some contexts, more appropriate. An invariant, as discussed here, is a practical – and therefore typically economical and logically incomplete – check on an object's state.

The notion of invariants has its origins in the work of Floyd, Naur, and Hoare on preconditions and postconditions and is present in essentially all work on abstract data types and program verification done over the last 30 years or so. It is also a staple of C debugging.

Typically, the invariant is not maintained during the execution of a member function. Functions that may be called while the invariant is invalid should not be part of the public interface. Private and protected functions can serve that purpose.

How can we express the notion of an invariant in a C++ program? A simple way is to define an invariant-checking function and insert calls to it in the public operations. For example:

```
class String {
        int sz;
        char* p;
public:
        class Range {};                         // exception classes
        class Invariant {};

        enum { TOO_LARGE = 16000 };             // length limit

        void check();                           // invariant check

        String(const char* q);
        String(const String&);
        ~String();

        char& operator[](int i);
        int size() { return sz; }

        // ...
};

void String::check()
{
        if (p==0 || sz<0 || TOO_LARGE<=sz || p[sz-1]) throw Invariant();
}
```

```
char& String::operator[](int i)
{
    check();                                    // check on entry
    if (i<0 || sz<=i) throw Range();            // do work
    check();                                    // check on exit
    return p[i];
}
```

This will work nicely and is hardly any work for the programmer. However, for a simple class like ***String*** the invariant checking will dominate the run time and maybe even the code size. Therefore, programmers often execute the invariant checks only during debugging:

```
inline void String::check()
{
#ifndef NDEBUG
    if (p==0 || sz<0 || TOO_LARGE<=sz || p[sz]) throw Invariant();
#endif
}
```

Here, the ***NDEBUG*** macro is used in a way similar to the way it is used in the standard C ***assert()*** macro. ***NDEBUG*** is conventionally set to indicate that debugging is *not* being done.

The simple act of defining invariants and using them during debugging is an invaluable help in getting the code right and – more importantly – in getting the concepts represented by the classes well defined and regular. The point is that when you are designing invariants, a class will be considered from an alternative viewpoint and the code will contain redundancy. Both increase the likelihood of spotting mistakes, inconsistencies, and oversights.

24.3.7.2 Assertions [lang.assert]

An invariant is a special form of an assertion. An assertion is simply a statement that a given logical criterion must hold. The question is what to do when it doesn't.

The C standard library – and by implication the C++ standard library – provides the ***assert()*** macro in ***<cassert>*** or ***<assert.h>***. An ***assert()*** evaluates its argument and calls ***abort()*** if the result is nonzero. For example:

```
void f(int* p)
{
    assert(p!=0);   // assert that p!=0; abort() if p is zero
    // ...
}
```

Before aborting, ***assert()*** outputs the name of its source file and the number of the line on which it appears. This makes ***assert()*** a useful debugging aid. ***NDEBUG*** is usually set by compiler options on a per-compilation-unit basis. This implies that ***assert()*** shouldn't be used in inline functions and template functions that are included in several translation units unless great care is taken that ***NDEBUG*** is set consistently (§9.2.3). Like all macro magic, this use of ***NDEBUG*** is too low-level, messy, and error-prone. Also, it is typically a good idea to leave at least some checks active in even the best-checked program, and ***NDEBUG*** isn't well suited for that. Furthermore, calling ***abort()*** is rarely acceptable in production code.

The alternative is to use an *Assert*( ) template that throws an exception rather than aborting so that assertions can be left in production code when that is desirable. Unfortunately, the standard library doesn't provide an *Assert*( ). However, it is trivially defined:

```
template<class X, class A> inline void Assert(A assertion)
{
    if (!assertion) throw X();
}
```

Assert( ) throws the exception *X*( ) if the *assertion* is false. For example:

```
class Bad_arg { };

void f(int* p)
{
    Assert<Bad_arg>(p!=0);  // assert p!=0; throw Bad_arg unless p!=0
    // ...
}
```

This style of assertion has the condition explicit, so if we want to check only while debugging we must say so. For example:

```
void f2(int* p)
{
    Assert<Bad_arg>(NDEBUG || p!=0);    // either I'm not debugging or p!=0
    // ...
}
```

The use of || rather than && in the assertion may appear surprising. However, *Assert<E>*(*a*||*b*) tests !(*a*||*b*) which is !*a*&&!*b*.

Using *NDEBUG* in this way requires that we define *NDEBUG* with a suitable value whether or not we are debugging. A C++ implementation does not do this for us by default, so it is better to use a value. For example:

```
#ifdef NDEBUG
const bool ARG_CHECK = false;          // we are not debugging: disable checks
#else
const bool ARG_CHECK = true;           // we are debugging
#endif

void f3(int* p)
{
    Assert<Bad_arg>(!ARG_CHECK || p!=0);    // either I'm not debugging or p!=0
    // ...
}
```

If the exception associated with an assertion is not caught, a failed *Assert*( ) *terminate*( )s the program much like an equivalent *assert*( ) would *abort*( ). However, an exception handler may be able to take some less drastic action.

In any realistically-sized program, I find myself turning assertions on and off in groups to suit the need for testing. Using *NDEBUG* is simply the crudest form of that technique. Early on in development, most assertions are enabled, whereas only key sanity checks are left enabled in

shipped code. This style of usage is most easily managed if the actual assertion is in two parts, with the first being an enabling condition (such as *ARG_CHECK*) and the second being the assertion proper.

If the enabling condition is a constant expression, the whole assertion will be compiled away when not enabled. However, the enabling condition can also be a variable so that it can be turned on and off at run time as debugging needs dictate. For example:

```
bool string_check = true;

inline void String::check()
{
    Assert<Invariant>(!string_check || (p && 0<=sz && sz<TOO_LARGE && p[sz]==0));
}

void f()
{
    String s = "wonder";
    // strings are checked here
    string_check = false;
    // no checking of strings here
}
```

Naturally, code will be generated in such cases, so we must keep an eye out for code bloat if we use such assertions extensively.

Saying

```
Assert<E>(a);
```

is simply another way of saying

```
if (!a) throw E();
```

Then why bother with *Assert*(), rather than writing out the statement directly? Using *Assert*() makes the designer's intent explicit. It says that this is an assertion of something that is supposed to be always true. It is not an ordinary part of the program logic. This is valuable information to a reader of the program. A more practical advantage is that it is easy to search for *assert*() or *Assert*() whereas searching for conditional statements that throw exceptions is nontrival.

Assert() can be generalized to throw exceptions taking arguments and variable exceptions:

```
template<class A, class E> inline void Assert(A assertion, E except)
{
    if (!assertion) throw except;
}

struct Bad_g_arg {
    int* p;
    Bad_g_arg(int* pp) :p(pp) { }
};

bool g_check = true;
int g_max = 100;
```

```
void g(int* p, exception e)
{
    Assert(!g_check || p!=0, e);                          // pointer is valid
    Assert(!g_check || (0<*p&&*p<=g_max), Bad_g_arg(p));   // value is plausible
    // ...
}
```

In many programs, it is crucial that no code is generated for an *Assert*( ) where the assertion can be evaluated at compile time. Unfortunately, some compilers are unable to achieve this for the generalized *Assert*( ). Consequently, the two-argument *Assert*( ) should be used only when the exception is not of the form *E*( ) and it is also acceptable for some code to be generated independently of the value of the assertion.

In §23.4.3.5, it was mentioned that the two most common forms of class hierarchy reorganizations were to split a class into two and to factor out the common part of two classes into a base class. In both cases, well-designed invariants can give a clue to the potential for reorganization. Comparing the invariant with the code of operations will show most of the invariant checking to be redundant in a class that is ripe for splitting. In such cases, subsets of the operations will access only subsets of the object state. Conversely, classes that are ripe for merging will have similar invariants even if their detailed implementations differ.

24.3.7.3 Preconditions and Postconditions [lang.pre]

One popular use of assertions is to express preconditions and postconditions of a function. That is, checking that basic assumptions about input hold and verifying that the function leaves the world in the expected state upon exit. Unfortunately, the assertions we would like to make are often at a higher level than the programming language allows us to express conveniently and efficiently. For example:

```
template<class Ran> void sort(Ran first, Ran last)
{
    Assert<Bad_sequence>("[first,last) is a valid sequence");      // pseudo code

    // ... sorting algorithm ...

    Assert<Failed_sort>("[first,last) is in increasing order");    // pseudo code
}
```

This problem is fundamental. What we want to say *about* a program is best expressed in a mathematically-based higher language, rather than in the algorithmic programming language *in which* we write the program.

As for invariants, a certain amount of cleverness is needed to translate the ideal of what we would like to assert into something that is algorithmically feasible to check. For example:

```
template<class Ran> void sort(Ran first, Ran last)
{
    // [first,last) is a valid sequence: check plausibility:
    Assert<Bad_sequence>(NDEBUG || first<=last);

    // ... sorting algorithm ...
```

```
        // [first,last) is in increasing order: check a sample:
        Assert<Failed_sort>(NDEBUG ||
                (last-first<2 || (*first<=last[-1]
                        && *first<=first[(last-first)/2] && first[(last-first)/2]<=last[-1])));
}
```

I often find writing ordinary code-checking arguments and results simpler than composing assertions. However, it is important to try to express the real (ideal) preconditions and postconditions – and at least document them as comments – before reducing them to something less abstract that can be effectively expressed in a programming language.

Precondition checking can easily degenerate into simple checking of argument values. As an argument is often passed through several functions, this checking can be repetitive and expensive. However, simply asserting that every pointer argument is nonzero in every function is not particularly helpful and can give a false sense of security – especially if the tests are done during debugging only to prevent overhead. This is a major reason why I recommend a focus on invariants.

24.3.7.4 Encapsulation [lang.encapsulate]

Note that in C++, the class – not the individual object – is the unit of encapsulation. For example:

```
class List {
        List* next;
public:
        bool on(List*);
        // ...
};

bool List::on(List* p)
{
        if (p == 0) return false;
        for(List* q = this; q; q=q->next) if (p == q) return true;
        return false;
}
```

The chasing of the private ***List::next*** pointer is accepted because ***List::on()*** has access to every object of class ***List*** it can somehow reference. Where that is inconvenient, matters can be simplified by not taking advantage of the ability to access the representation of other objects from a member function. For example:

```
bool List::on(List* p)
{
        if (p == 0) return false;
        if (p == this) return true;
        if (next==0) return false;
        return next->on(p);
}
```

However, this turns iteration into recursion, and doing that can cause a major performance hit when a compiler isn't able to optimize the recursion back into an iteration.

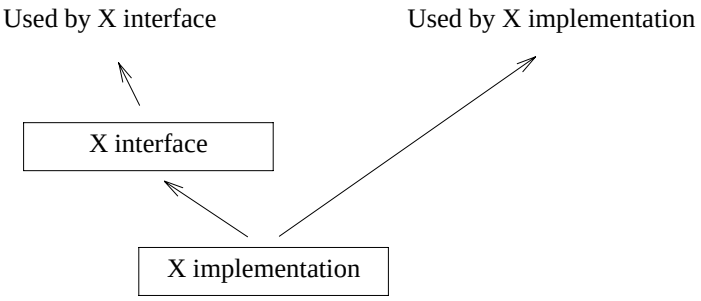
24.4 Components [lang.component]

The unit of design is a collection of classes, functions, etc., rather than an individual class. Such a collection, often called a *library* or a *framework* (§25.8), is also the unit of reuse (§23.5.1), maintenance, etc. C++ provides three mechanisms for expressing the notion of a set of facilities united by a logical criteria:

[1] A class – containing a collection of data, function, template, and type members
[2] A class hierarchy – containing a collection of classes
[3] A namespace – containing a collection of data, function, template, and type members

A class provides many facilities to make it convenient to create objects of the type it defines. However, many significant components are not best described by a mechanism for creating objects of a single type. A class hierarchy expresses the notion of a set of related types. However, the individual members of a component are not always best expressed as classes and not all classes possess the basic similarity required to fit into a meaningful class hierarchy (§24.2.5). Therefore, a namespace is the most direct and the most general embodiment of the notion of a component in C++. A component is sometimes referred to as a ''class category.'' However, not every element of a component is or should be a class.

Ideally, a component is described by the set of interfaces it uses for its implementation plus the set of interfaces it provides for its users. Everything else is ''implementation detail'' and hidden from the rest of the system. This may indeed be the designer's description of a component. To make it real, the programmer needs to map it into declarations. Classes and class hierarchies provide the interfaces, and namespaces allow the programmer to group the interfaces and to separate interfaces used from interfaces provided. Consider:



Using the techniques described in §8.2.4.1, this becomes:

```
namespace A { // some facilities used by X's interface
    // ...
}

namespace X { // interface of component X

    using namespace A; // dependent on declarations from A
    // ...
    void f();
}
```

```
namespace X_impl { // facilities needed by X's implementation
    using namespace X;
    // ...
}

void X::f()
{
    using namespace X_impl; // dependent on declarations from X_impl
    // ...
}
```

The general interface ***X*** should not depend on the implementation interface ***X_impl***.

A component can have many classes that are not intended for general use. Such classes should be ''hidden'' within implementation classes or namespaces:

```
namespace X_impl { // component X implementation details

    class Widget {
        // ...
    };

    // ...
}
```

This ensures that ***Widget*** isn't used from other parts of the program. However, classes that represent coherent concepts are often candidates for reuse and should therefore be considered for inclusion into the interface of the component. Consider:

```
class Car {
    class Wheel {
        // ...
    };

    Wheel flw, frw, rlw, rrw;
    // ...
public:
    // ...
};
```

In most contexts, we need to have the actual wheels hidden to maintain the abstraction of a car (when you use a car you cannot operate the wheels independently). However, the ***Wheel*** class itself seems a good candidate for wider use, so moving it outside class ***Car*** might be better:

```
class Wheel {
    // ...
};

class Car {
    Wheel flw, frw, rlw, rrw;
    // ...
public:
    // ...
};
```

The decision to nest or not depends on the aims of the design and the generality of the concepts involved. Both nesting and ''non-nesting'' are widely applicable techniques for expressing a design. The default should be to make a class as local as possible until a need to make it more generally available is demonstrated.

There is a nasty tendency for ''interesting'' functions and data to ''bubble up'' to the global namespace, to widely-used namespaces, or to ultimate base classes in a hierarchy. This can easily lead to unintentional exposure of implementation details and to the problems associated with global data and global functions. This is most likely to happen in a single-rooted hierarchy, and in a program where only very few namespaces are used. Virtual base classes (§15.2.4) can be used to combat this phenomenon in the context of class hierarchies. Small ''implementation'' namespaces are the main tool for avoiding the problem in the context of namespaces.

Note that header files provide a powerful mechanism for supplying different views of a component to different users and for excluding classes that are considered part of the implementation from the user's view (§9.3.2).

24.4.1 Templates [lang.temp]

From a design perspective, templates serve two, weakly-related needs:

- Generic programming
- Policy parameterization

Early in a design effort, operations are just operations. Later, when it is time to specify the type of operands templates become essential when using a statically-typed programming language, such as C++. Without templates, function definitions would have to be replicated or checking would have to be unnecessarily postponed to run time (§24.2.3). An operation that implements an algorithm for a variety of operand types is a candidate to be implemented as a template. If all operands fit into a single class hierarchy, and especially if there is a need to add new operand types at run time, the operand type is best represented as a class – often as an abstract class. If the operand types do not fit into a single hierarchy and especially if run-time performance is critical, the operation is best implemented as a template. The standard containers and their supporting algorithms are an example of when the need to take operands of a variety of unrelated types combined with a need for run-time performance lead to the use of templates (§16.2).

To make the template/hierarchy tradeoff more concrete, consider how to generalize a simple iteration:

```
void print_all(Iter_for_T x)
{
    for (T* p = x.first(); p; p = x.next()) cout << *p;
}
```

Here, the assumption is that *Iter_for_T* provides operations that yield *T**s.

We can make the iterator *Iter_for_T* a template parameter:

```
template<class Iter_for_T> print_all(Iter_for_T x)
{
    for (T* p = x.first(); p; p = x.next()) cout << *p;
}
```

This allows us to use a variety of unrelated iterators as long as they all provide *first*() and *next*() with the right meanings and as long as we know the type of iterator for each call of *print_all*() at compile time. The standard library containers and algorithms are based on this idea.

Alternatively, we can use the observation that *first*() and *next*() constitute an interface to iterators. We can then define a class to represent that interface:

```
class Iter {
public:
        virtual T* first() const = 0;
        virtual T* next() = 0;
};

void print_all2(Iter& x)
{
        for (T* p = x.first(); p; p = x.next()) cout << *p;
}
```

We can now use every iterator derived from *Iter*. The actual code doesn't differ depending on whether we use templates or a class hierarchy to represent the parameterization – only the run-time, recompilation, etc., tradeoffs differ. In particular, class *Iter* is a candidate for use as an argument for the template:

```
void f(Iter& i)
{
        print_all(i);     // use the template
        print_all2(i);
}
```

Consequently, the two approaches can be seen as complementary.

Often, a template needs to use functions and classes as part of its implementation. Many of those must themselves be templates so as to maintain generality and efficiency. In that way, algorithms become generic over a range of types. This style of template use is called *generic programming* (§2.7). When we call *std*::*sort*() on a *vector*, the elements of the vector are the operands of the *sort*(); thus, *sort*() is generic for the element types. In addition, the standard sort is generic for the container types because it is invoked on iterators for arbitrary, standard-conforming containers (§16.3.1).

The *sort*() algorithm is also parameterized on the comparison criteria (§18.7.1). From a design perspective, this is different from taking an operation and making it generic on its operand type. Deciding to parameterize an algorithm on an object (or operation) in a way that controls the way the algorithm operates is a much higher-level design decision. It is a decision to give the designer/programmer control over some part of the policy governing the operation of the algorithm. From a programming language point of view, however, there is no difference.

24.4.2 Interfaces and Implementations [lang.interface]

The ideal interface

- presents a complete and coherent set of concepts to a user,
- is consistent over all parts of a component,

– does not reveal implementation details to a user,
– can be implemented in several ways,
– is statically typed,
– is expressed using application-level types, and
– depends in limited and well-defined ways on other interfaces.

Having noted the need for consistency across the classes that present the component's interface to the rest of the world (§24.4), we can simplify the discussion by looking at only a single class. Consider:

```
class Y { /* ... */ };        // needed by X

class Z { /* ... */ };        // needed by X

class X {  // example of poor interface style
        Y a;
        Z b;
public:
        void f(const char * ...);
        void g(int[], int);
        void set_a(Y&);
        Y& get_a();
};
```

This interface has several potential problems:

– The interface uses the types ***Y*** and ***Z*** in a way that requires the declarations of ***Y*** and ***Z*** to be known to compile it.
– The function ***X*****::*****f*****()** takes an arbitrary number of arguments of unknown types (probably somehow controlled by a ''format string'' supplied as the first argument; §21.8).
– The function ***X*****::*****g*****()** takes an ***int*****[]** argument. This may be acceptable, but typically it is a sign that the level of abstraction is too low. An array of integers is not self-describing, so it is not obvious how many elements it is supposed to have.
– The ***set_a*****()** and ***get_a*****()** functions most likely expose the representation of objects of class ***X*** by allowing direct access to ***X*****::*****a***.

These member functions provide an interface at a very low level of abstraction. Basically, classes with interfaces at this level belong among the implementation details of a larger component – if they belong anywhere at all. Ideally, an argument of an interface function carries enough information to make it self-describing. A rule of thumb is that it should be possible to transmit the request for service over a thin wire for service at a remote server.

C++ allows the programmer to expose the representation of a class as part of the interface. This representation may be hidden (using ***private*** or ***protected***), but it is available to the compiler to allow allocation of automatic variables, to allow inline substitution of functions, etc. The negative effect of this is that use of class types in the representation of a class may introduce undesirable dependencies. Whether the use of members of types ***Y*** and ***Z*** is a problem depends on what kind of types ***Y*** and ***Z*** actually are. If they are simple types, such as ***list***, ***complex***, and ***string***, their use is most often quite appropriate. Such types can be considered stable, and the need to include their class declarations is an acceptable burden on the compiler. However, if ***Y*** and ***Z*** themselves had been interface classes of significant components, such as a graphics system or a bank account

management system, it might be wise not to depend too directly on them. In such cases, using a pointer or a reference member is often a better choice:

```
class Y;
class Z;

class X { // X accesses Y and Z through pointers and references only
      Y* a;
      Z& b;
      // ...
};
```

This decouples the definition of ***X*** from the definitions of ***Y*** and ***Z***; that is, the definition of ***X*** depends on the names ***Y*** and ***Z*** only. The implementation of ***X*** will, of course, still depend on the definitions of ***Y*** and ***Z***, but this will not adversely affect the users of ***X***.

This illustrates an important point: an interface that hides significant amounts of information – as a useful interface ought to – will have far fewer dependencies than the implementation it hides. For example, the definition of class ***X*** can be compiled without access to the definitions of ***Y*** and ***Z***. However, the definitions of ***X***'s member functions that manipulate the ***Y*** and ***Z*** objects will need access to the definitions of ***Y*** and ***Z***. When dependencies are analyzed, the dependencies of the interface and the implementation must be considered separately. In both cases, the ideal is for the dependency graphs of a system to be directed acyclic graphs to ease understanding and testing of the system. However, this ideal is far more critical and far more often achievable for interfaces than for implementations.

Note that a class can define three interfaces:

```
class X {
private:
      // accessible to members and friends only
protected:
      // accessible to members and friends and
      // to members and friends of derived classes only
public:
      // accessible to the general public
};
```

In addition, a ***friend*** is part of the public interface (§11.5).

A member should be part of the most restrictive interface possible. That is, a member should be ***private*** unless there is a reason for it to be more accessible. If it needs to be more accessible, it should be ***protected*** unless there is a reason for it to be ***public***. It is almost always a bad idea to make a data member ***public*** or ***protected***. The functions and classes that constitute the public interface should present a view of the class that fits with its role as representing a concept.

Note that abstract classes can be used to provide a further level of representation hiding (§2.5.4, §12.3, §25.3).

24.4.3 Fat Interfaces [lang.fat]

Ideally, an interface should offer only operations that make sense and that can be implemented well by every derived class implementing that interface. However, this is not always easy. Consider lists, arrays, associative arrays, trees, etc. As shown in §16.2.2, it is tempting and sometimes useful to provide a generalization of all of these types – usually called a *container* – that can be used as the interface to every one of these. This (apparently) relieves the user of having to deal with the details of all of these containers. However, defining the interface of a general container class is nontrivial. Assume that we want to define ***Container*** as an abstract type. What operations do we want ***Container*** to provide? We could provide only the operations that every container can support – the intersection of the sets of operations – but that is a ridiculously narrow interface. In fact, in many interesting cases that intersection is empty. Alternatively, we could provide the union of all the sets of operations and give a run-time error if a ''non-existent'' operation is applied to an object through this interface. An interface that is such a union of interfaces to a set of concepts is called a *fat interface*. Consider a ''general container'' of objects of type ***T***:

```
class Container {
public:
    struct Bad_oper {    // exception class
        const char* p;
        Bad_oper(const char* pp) : p(pp) { }
    };

    virtual void put(const T*) { throw Bad_oper("Container::put"); }
    virtual T* get() { throw Bad_oper("Container::get"); }

    virtual T*& operator[](int) { throw Bad_oper("Container::[](int)"); }
    virtual T*& operator[](const char*) { throw Bad_oper("Container::[](char*)"); }
    // ...
};
```

*Container*s could then be declared like this:

```
class List_container : public Container, private list {
public:
    void put(const T*);
    T* get();
    // ... no operator[] ...
};

class Vector_container : public Container, private vector {
public:
    T*& operator[](int);
    T*& operator[](const char*);
    // ... no put() or get() ...
};
```

As long as one is careful, all is well:

```
void f()
{
    List_container sc;
    Vector_container vc;
    // ...
    user(sc,vc);
}

void user(Container& c1, Container& c2)
{
    T* p1 = c1.get();
    T* p2 = c2[3];
    // don't use c2.get() or c1[3]
    // ...
}
```

However, few data structures support both the subscripting and the list-style operations well. Consequently, it is probably not a good idea to specify an interface that requires both. Doing so leads to the use of run-time type-inquiry (§15.4) or exception handling (Chapter 14) to avoid run-time errors. For example:

```
void user2(Container& c1, Container& c2) // detection is easy, but recovery can be hard
{
    try {
        T* p1 = c1.get();
        T* p2 = c2[3];
        // ...
    }
    catch(Container::Bad_oper& bad) {
        // Oops!
        // Now what?
    }
}
```

or

```
void user3(Container& c1, Container& c2) // early detection is tedious; recovery can still be hard
{
    if (dynamic_cast<List_container*>(&c1) && dynamic_cast<Vector_container*>(&c2)) {
        T* p1 = c1.get();
        T* p2 = c2[3];
        // ...
    }
    else {
        // Oops!
        // Now what?
    }
}
```

In both cases, run-time performance can suffer and the generated code can be surprisingly large. As a result, people are tempted to ignore the potential errors and hope that they don't actually occur

when the program is in the hands of users. The problem with this approach is that exhaustive testing is also hard and expensive.

Consequently, fat interfaces are best avoided where run-time performance is at a premium, where strong guarantees about the correctness of code are required, and in general wherever there is a good alternative. The use of fat interfaces weakens the correspondence between concepts and classes and thus opens the floodgates for the use of derivation as a mere implementation convenience.

24.5 Advice [lang.advice]

[1] Evolve use towards data abstraction and object-oriented programming; §24.2.
[2] Use C++ features and techniques as needed (only); §24.2.
[3] Match design and programming styles; §24.2.1.
[4] Use classes/concepts as a primary focus for design rather than functions/processing; §24.2.1.
[5] Use classes to represent concepts; §24.2.1, §24.3.
[6] Use inheritance to represent hierarchical relationships between concepts (only); §24.2.2, §24.2.5, §24.3.2.
[7] Express strong guarantees about interfaces in terms of application-level static types; §24.2.3.
[8] Use program generators and direct-manipulation tools to ease well-defined tasks; §24.2.4.
[9] Avoid program generators and direct-manipulation tools that do not interface cleanly with a general-purpose programming language; §24.2.4.
[10] Keep distinct levels of abstraction distinct; §24.3.1.
[11] Focus on component design; §24.4.
[12] Make sure that a virtual function has a well-defined meaning and that every overriding function implements a version of that desired behavior; §24.3.4, §24.3.2.1.
[13] Use public inheritance to represent *is-a* relationships; §24.3.4.
[14] Use membership to represent *has-a* relationships; §24.3.4.
[15] Prefer direct membership over a pointer to a separately-allocated object for expressing simple containment; §24.3.3, §24.3.4.
[16] Make sure that the *uses* dependencies are understood, non-cyclic wherever possible, and minimal; §24.3.5.
[17] Define invariants for all classes; §24.3.7.1.
[18] Explicitly express preconditions, postconditions, and other assertions as assertions (possibly using ***Assert***`()`); §24.3.7.2.
[19] Define interfaces to reveal the minimal amount of information needed; §24.4.
[20] Minimize an interface's dependencies on other interfaces; §24.4.2.
[21] Keep interfaces strongly typed; §24.4.2.
[22] Express interfaces in terms of application-level types; §24.4.2.
[23] Express an interface so that a request could be transmitted to a remote server; §24.4.2.
[24] Avoid fat interfaces; §24.4.3.
[25] Use ***private*** data and member functions wherever possible; §24.4.2.
[26] Use the ***public/protected*** distinction to distinguish between the needs of designers of derived classes and general users; §24.4.2.

[27] Use templates for generic programming; §24.4.1.
[28] Use templates to parameterize an algorithm by a policy; §24.4.1.
[29] Use templates where compile-time type resolution is needed; §24.4.1.
[30] Use class hierarchies where run-time type resolution is needed; §24.4.1.