C++??

A Critique of C++

and Programming and Language Trends of the 1990s

3rd Edition

Ian Joyner

1. Introduction

This is now the third edition of this critique; it has been four years since the last edition. The main factor to precipitate a new edition is that there are now more environments and languages available that rectify the problems of C++. The last edition was addressed to people who were considering adopting C++, in particular managers who would have to fund projects. There are now more choices, so comparison to the alternatives makes the critique less hypothetical. The critique was not meant as an academic treatise, although some of the aspects relating to inheritance, etc., required a bit of technical knowledge.

The critique is long; it would be good if it were shorter, but that would be possible only if there were less flaws in C++. Even so, the critique is not exhaustive of the flaws: I find new traps all the time. Instead of documenting every trap, the critique attempts to arrange the traps into categories and principles. This is because the traps are not just one off things, but more deeply rooted in the principles of C++. Neither is the critique a repository of ‘guess what this obscure code does’ examples.

One desired outcome of this critique is that it should awaken the industry about the C++ myth and the fact that there are now viable alternatives to C++ that do not suffer from as many technical problems. The industry needs less hype and more sensible programming practices. No language can be perfect in every situation, and tradeoffs are sometimes necessary, but you can now feel freer to choose a language which is more closely suited to your needs. The alternatives to C++ provide no *silver bullet*, but significantly reduce the risks and costs of software development compared to C++. The alternatives do not suffer under the complexities of C++ and do not burden the programmer with many trivialities which the compiler should handle; and they avoid many of the flaws and inanities of C/C++.

The language events which have made an update desirable are the introduction of Java, the wider availability of more stable versions of Eiffel, and the finalisation of the Ada 95 standard. Java in particular set out to correct the flaws of C++, and most sections in the original critique now make some comment on how Java addresses the problems. Eiffel never did have the same flaws as C++, and has been around since long before the original critique. Eiffel was designed to be object-oriented from the ground up, rather than a *bolt-on*. Java offers better integration with OO than C++. Now that there are language comparisons in the critique the arguments are less hypothetical, and the criticisms of C++ are more concrete.

Another factor has been the publishing of Bjarne Stroustrup’s “Design and Evolution of C++” [Stroustrup 94]. This has many explanations of the problems of extending C with object-oriented extensions while retaining *compatibility* with C. In many ways, Stroustrup reinforces comments that I made in the original critique, but I differ from Stroustrup in that I do not view the flaws of C++ as acceptable, even if they are widely known, and many programmers know how to avoid the traps. Programming is a complex endeavour: complex and flawed languages do not help.

A question which has been on my mind in the last few years is when is OO applicable? OO is a universal paradigm. It is very general and powerful. There is nothing that you could not program in it. But is this always appropriate? Lower level programmers have tended to keep writing such things as device drivers in C. It is not lower levels that I am interested in, but the higher levels. OO might still be too low level for a number of applications. A recent book [Shaw 96] suggests that software engineers are too busy designing systems in terms of stacks, lists, queues, etc., instead of adopting higher level, domain-oriented architectures. [Shaw 96] offers some hope to the industry that we are learning how to architect to solve problems, rather than distorting problems to fit particular technologies and solutions.

For instance, commercial and business programming might be faster using a paradigm involving business objects. While these could be provided in an OO framework, the generality is not needed in commercial processing, and will slow and limit the flexibility of the development process. By analogy, walking is a fine mode of transport, but do I choose to walk everywhere? There seems to be a potentially large market for specialised paradigms, which support rapid application development (RAD) techniques. These paradigms may be based on some OO language, framework and libraries in the background. In anything though, we should be cautious, as this is an industry particularly prone to buzzwords and fads.

The second edition generated a lot of interest, and it was published in a number of places: Software Design in Japan translated it into Japanese, and published it over a series of months in 1993; it was published in an abridged form in TOOLS Pacific 1992; it was also published in Gregory’s A Series Technical Journal. However, I resisted handing over copyright to anyone, as I wanted the paper to be freely available on the Internet; it is now available on more sites than I know about. My thanks to all those who have been so supportive of the 2nd edition.

Another reason for the 3rd edition is that the original critique was very much a product of newsgroup discussions. In this edition, I have attempted to at least improve the readability and flow, while not changing the overall structure or embarking on a complete rewrite. The primary goal has been to annotate the original with comparisons to Java and Eiffel.

C++ has become even more widely used over the last few years. However, people are starting to realise that it is not the answer to all programming problems, or that retaining compatibility with C is a good thing. In some sectors there has been a backlash, precipitated by the fact that people have found the production of defect free quality software an extremely difficult and costly task. OO has been over-hyped, but neither are its real benefits present in C++.

It is important and timely to question C++'s success. Several books are already published on the subject [Sakkinen 92], [Yoshida 92], and [Wiener 95]. A paper on the recommended practices for use in C++ [Ellemtel 92] suggests "C++ is a difficult language in which there may be a very fine line between a feature and a bug. This places a large responsibility upon the programmer." Is this a responsibility or a burden? The 'fine line' is a result of an unnecessarily complicated language definition. The C++ standardisation committee warns "C++ is already too large and complicated for our taste" [X3J16 92].

Sun's Java White Paper [Sun 95] says that in designing Java, "The first step was to *eliminate redundancy* from C and C++. In many ways, the C language evolved into a collection of overlapping features, providing too many ways to do the same thing, while in many cases not providing needed features. C++, even in an attempt to add "classes in C" merely added more redundancy while retaining the inherent problems of C."

The designer of Eiffel, Bertrand Meyer, states in the appendix "On language design and evolution" in [Meyer 92] some guiding principles of language design: simplicity vs complexity, uniqueness, consistency. "The Principle of Uniqueness," Meyer says, "is easily expressed: the language should provide one good way to express every operation of interest; it should avoid providing two."

Meyer has produced a seminal work on OO: *Object-oriented Software Construction*, [Meyer 88]. All software engineers and object-oriented practitioners should read and absorb this work. A completely revised 2nd edition is soon to appear. A later short book "Object Success" is directed to managers (probably the reason for the pun in the name), with an overview of OO, [Meyer 95].

While C programmers can immediately use C++ to write and compile C programs, this does not take advantage of OO. Many see this as a strength, but it is often stated that the C base is C++'s greatest weakness. However, C++ adds its own layers of complexity, like its handling of multiple inheritance, overloading, and others. I am not so sure that C is C++'s greatest weakness. Java has shown that in removing C constructs that do not fit with object-oriented concepts, that C can provide an acceptable, albeit not perfect base.

Adoption of C++ does not suddenly transform C programmers into object-oriented programmers. A complete change of thinking is required, and C++ actually makes this difficult. A critique of C++ cannot be separated from criticism of the C base language, as it is essential for the C++ programmer to be fluent in C. Many of C's problems affect the way that object-orientation is implemented and used in C++. This critique is not exhaustive of the weaknesses of C++, but it illustrates the practical consequences of these weaknesses with respect to the timely and economic production of quality software.

This paper is structured as follows: section 2 considers the role of a programming language; section 3 examines some specific aspects of C++; section 4 looks specifically at C; and the conclusion examines where C++ has left us, and considers the future.

I have tried to keep the sections reasonably self contained, so that you can read the sections that interest you, and use the critique in a reference style. There are some threads that occur throughout the critique, and you will find some repetition of ideas to achieve self contained sections.

Having said that, I hope that you find this critique useful, and enjoyable: so please feel free to distribute it to your management, peers and friends.

2. The Role of a Programming Language

A programming language functions at many different levels and has many roles, and should be evaluated with respect to those levels and roles. Historically, programming languages have had a limited role, that of writing executable programs. As programs have grown in complexity, this role alone has proved insufficient. Many design and analysis techniques have arisen to support other necessary roles.

Object-oriented techniques help in the analysis and design phases; object-oriented languages to support the implementation phase of OO, but in many cases these lack uniformity of concepts, integration with the development environment and commonality of purpose. Traditional problematic software practices are infiltrating the object-oriented world with little thought. Often these techniques appeal to management because they are outwardly organised: people are assigned organisational roles such as project manager, team leader, analyst, designer and programmer. But these techniques are simplistic and insufficient, and result in demotivated and uncreative environments.

Object-orientation, however, offers a better rational approach to software development. The complementary roles of analysis, design, implementation and project organisation should be better integrated in the object-oriented scheme. This results in economical software production, and more creative and motivated environments.

The organisation of projects also required tools external to the language and compiler, like 'make.' A re-evaluation of these tools shows that often the division of labour between them has not been done along optimal lines: firstly, programmers need to do extra *bookkeeping* work which could be automated; and secondly, inadequate *separation of concerns* has resulted in inflexible software systems.

C++ is an interesting experiment in adapting the advantages of object-orientation to a traditional programming language and development environment. Bjarne Stroustrup should be recognised for having the insight to put the two technologies together; he ventured into OO not only before solutions were known to many issues, but before the issues were even widely recognised. He deserves better than a back full of arrows. But in retrospect, we now treat concepts such as multiple inheritance with a good deal of respect, and realise that the Unix development environment with limited linker support does not provide enough compiler support for many of the features that should be in a high level language.

There are solutions to the problems that C++ uncovered. C++ has gone down a path in research, but now we know what the problems are and how to solve them. Let's adopt or develop such languages. Fortunately, such languages have been developed, which are of industrial strength, meant for commercial projects, and are not just academic research projects. It is now up to the industry to adopt them on a wider scale.

C++, however, retains the problems of the old order of software production. C++ has an advantage over C as it supports many facets of object-orientation. These can be used for some analysis and design. The processes of analysis, design, and organisation, however, are still largely external to C++. C++ has not realised the important advantages of integrated software development that leads to improved economies of software production.

Java is an interesting development taking a different approach to C++: strict compatibility with C is not seen as a relevant goal. Java is not the only C based alternative to C++ in the object-oriented world. There has also been Objective-C from Brad Cox, and mainly used in NeXT's OpenStep environment. Objective-C is more like Smalltalk, in that all binding is done dynamically at run time.

A language should not only be evaluated from a technical point of view, considering its syntactic and semantic features; it should also be analysed from the viewpoint of its contribution to the entire software development process. A language should enable communication between project members acting at different levels, from management, who set enterprise level policies, to testers, who must test the result. All these people are involved in the general activity of programming, so a language should enable communication between project members separated in space and time. A single programmer is not often responsible for a task over its entire lifetime.

2.1 Programming

Programming and specification are now seen as the same task. One man's specification is another's program. Eventually you get to the point of processing a specification with a compiler, which generates a program which actually runs on a computer. Carroll Morgan banishes the distinction between specifications and programs: "To us they are **all** programs." [Morgan 90]. Programming is a term that not only refers to implementation; programming refers to the whole process of analysis, design and implementation.

The Eiffel language integrates the concept of specification and programming, rejecting the divided models of the past in favour of a new integrated approach to projects. Eiffel achieves this in several ways: it has a clean clear syntax which is easy to read, even by non-programmers; it has techniques such as preconditions and postconditions so that the semantics of a routine can be clearly documented, these being borrowed from formal specification techniques, but made easy for the 'rest of us' to use; and it has tools to extract the abstract specification from the implementation details of a program. Thus Eiffel is more than just a language, providing a whole integrated development environment.

Chris Reade [Reade 89] gives the following explanation of programming and languages. "One, rather narrow, view is that a **program** is a sequence of instructions for a machine. We hope to show that there is much to be gained from taking the much broader view that programs are descriptions of values, properties, methods, problems and solutions. The role of the machine is to speed up the manipulation of these descriptions to provide solutions to particular problems. A **programming language** is a convention for writing descriptions which can be evaluated."

[Reade 89] also describes programming as being a "Separation of concerns". He says:

"The programmer is having to do several things at the same time, namely,

(1) describe what is to be computed;
(2) organise the computation sequencing into small steps;
(3) organise memory management during the computation."

Reade continues, "Ideally, the programmer should be able to concentrate on the first of the three tasks (describing what is to be computed) without being distracted by the other two, more administrative, tasks. Clearly, administration is important but by separating it from the main task we are likely to get more reliable results and we can ease the programming problem by automating much of the administration.

"The separation of concerns has other advantages as well. For example, program proving becomes much more feasible when details of sequencing and memory management are absent from the program. Furthermore, descriptions of what is to be computed should be free of such detailed step-by-step descriptions of how to do it if they are to be evaluated with different machine architectures. Sequences of small changes to a data object held in a store may be an inappropriate description of how to compute something when a highly parallel machine is being used with thousands of processors distributed throughout the machine and local rather than global storage facilities.

"Automating the administrative aspects means that the language implementor has to deal with them, but he/she has far more opportunity to make use of very different computation mechanisms with different machine architectures."

These quotes from Reade are a good summary of the principles from which I criticise C++. What Reade calls administrative tasks, I call *bookkeeping*. Bookkeeping adds to the cost of software production, and reduces flexibility which in turn adds more to the cost. C and C++ are often criticised for being cryptic. The reason is that C concentrates on points 2 and 3, while the description of what is to be computed is obscured.

High level languages describe 'what' is to be computed; that is the problem domain. 'How' a computation is achieved is in the low-level machine-oriented deployment domain. Automating the bookkeeping tasks enhances correctness, compatibility, portability and efficiency. Bookkeeping tasks arise from having to specify 'how' a computation is done. Specifying 'how' things are done in one environment hinders portability to other platforms.

The most significant way high level languages replace bookkeeping is using a declarative approach, whereas low level languages use operators, which make them more like assemblers. C and C++ provide operators rather than the declarative approach, so are low level. The declarative approach centralises decisions and lets the compiler generate the underlying machine operators. With the operator approach, the bookkeeping is on the programmer to use the correct operator to access an entity, and if a decision changes, the programmer will have to change all operators, rather than change the single declaration and simply recompiling. Thus in C and C++ the programmer is often concerned with the access mechanisms to data, whereas high level languages hide the implementation detail, making program development and maintenance far more flexible.

While C and C++ syntax is similar to high level language syntax, C and C++ cannot be considered high level, as they do not remove bookkeeping from the programmer that high level languages should, requiring the compiler to take care of these details. The low level nature of C and C++ severely impacts the development process.

The most important quality of a high level language is to remove bookkeeping burden from the programmer in order to enhance speed of development, maintainability and flexibility. This attribute is more important than object-orientation itself, and should be intrinsic to any modern programming paradigm. C++ more than cancels the benefits of OO by requiring programmers to perform much of the bookkeeping instead of it being automated.

The industry should be moving towards these ideals, which will help in the economic production of software, rather than the costly techniques of today. We should consider what we need, and assess the problems of what we have against that. Object-orientation provides one solution to these problems. The effectiveness of OO, however, depends on the quality of its implementation.

2.2 Communication, abstraction and precision

The primary purpose of any language is communication. A specification is communication from one person to another entity of a task to be fulfilled. At the lowest level, the task to be fulfilled is the execution of a program by a computer. At the next level it is the compilation of a program by a compiler. At higher levels, specifications communicate to other people what is to be accomplished by the programming task. At the lowest level, instructions must be precisely executed, but there is no understanding; it is purely mechanical. At higher levels, understanding is important, as human intelligence is involved, which is why enlightened management practices emphasise training rather than forced processes. This is not to say that precision is not important; precision at the higher levels is of utmost importance, or the rest of the endeavour will fail. Most projects fail due to lack of precision in the requirements and other early stages.

Unfortunately, often those who are least skilled in programming work at the higher levels, so specifications lack the desirable properties of abstraction and precision. Just as in the *Dilbert Principle* [Adams 96], the least effective programmers are promoted to where they will seemingly do the least damage. This is not quite the winning strategy that it seems, as that is where they actually do the most damage, as teams of confused programmers are then left to straighten out their specifications, while the so called analysts move onto the next project or company to sew the seeds of disaster there.

(Indeed, since many managers have not read or understood the works of Deming [Deming 82], [L&S 95], De Marco and Lister [DM&L 87], and Tom Peters' later works, the message that the physical environment and attitudes of the work place leads to quality has not got through. Perhaps the humour of Scott Adams is now the only way this message will have impact.)

At higher levels, abstraction facilitates understanding. Abstraction and precision are both important qualities of high level specifications. Abstraction does not mean vagueness, nor the abandonment of precision. Abstraction means the removal of irrelevant detail from a certain viewpoint. With an abstract specification, you are left with a precise specification; precisely the properties of the system that are relevant.

Abstraction is a fundamental concept in computing. Aho and Ullman say “An important part of the field [computer science] deals with how to make programming easier and software more reliable. But fundamentally, computer science is a science of *abstraction* -- creating the right model for a problem and devising the appropriate mechanizable techniques to solve it.” [Aho 92]. They also say “Abstraction in the sense we use it often implies simplification, the replacement of a complex and detailed real-world situation by an understandable model within which we can solve the problem.”

A well known example that exhibits both abstraction and precision is the London Underground map designed by Harold Beck. This is a diagrammatic map that has abstracted irrelevant details from the real London geography to result in a conveniently sized and more readable map. Yet the map precisely shows the underground stations and where passengers can change trains. Many other city transport systems have adopted the principles of Beck’s map. Using this model passengers can easily solve such problems as “How do I get from Knightsbridge to Baker Street?”

2.3 Notation

A programming language should support the exchange of ideas, intentions, and decisions between project members; it should provide a formal, yet readable, notation to support consistent descriptions of systems that satisfy the requirements of diverse problems. A language should also provide methods for automated project tracking. This ensures that modules (classes and functionality) that satisfy project requirements are completed in a timely and economic fashion. A programming language aids reasoning about the design, implementation, extension, correction, and optimisation of a system.

During requirements analysis and design phases, formal and semi-formal notations are desirable. Notations used in analysis, design, and implementation phases should be complementary, rather than contradictory. Currently, analysis, design and modelling notations are too far removed from implementation, while programming languages are in general too low level. Both designers and programmers must compromise to fill the gap. Many current notations provide difficult transition paths between stages. This ‘semantic gap’ contributes to errors and omissions between the requirements, design and implementation phases.

Better programming languages are an implementation extension of the high level notations used for requirements analysis and design, which will lead to improved consistency between analysis, design and implementation. Object-oriented techniques emphasise the importance of this, as abstract definition and concrete implementation can be separate, yet provided in the same notation.

Programming languages also provide notations to formally document a system. Program source is the only reliable documentation of a system, so a language should explicitly support documentation, not just in the form of comments. As with all language, the effectiveness of communication is dependent upon the skill of the writer. Good program writers require languages that support the role of documentation, and that the language notation is perspicuous, and easy to learn. Those not trained in the skill of ‘writing’ programs, can read them to gain understanding of the system. After all, it is not necessary for newspaper readers to be journalists.

2.4 Tool Integration

A language definition should enable the development of integrated automated tools to support software development. For example, browsers, editors and debuggers. The compiler is just another tool, having a twofold role. Firstly, code generation for the target machine. The role of the machine is to execute the produced programs. A compiler has to check that a program conforms to the language syntax and grammar, so it can ‘understand’ the program in order to translate it into an executable form. Secondly, and more importantly, the compiler should check that the programmers expression of the system is valid, complete, and consistent; ie., perform semantics checks that a program is internally consistent. Generating a system that has detectable inconsistencies is pointless.

2.5 Correctness

Deciding what constitutes an inconsistency and how to detect it often raises passionate debate. The discord arises because the detectable inconsistencies do not exactly match real inconsistencies. There are two opposing views: firstly, languages that overcompensate are restrictive, you should trust your programmers; secondly, that programmers are human and make mistakes and program crashes at run-time are intolerable.

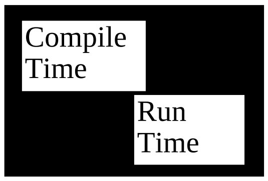
This is the key to the following diagrams:

■ Real Inconsistencies $\rightarrow$ Obscure failures

□ False Alarms

▨ Superfluous run-time checks/inefficiency

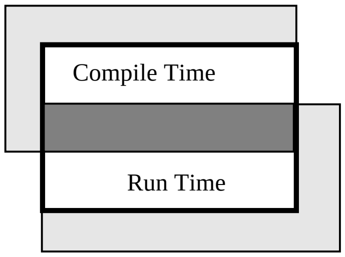
In the first figure the black box represents the real inconsistencies, which must be covered by either compile-time checks or run-time checks.



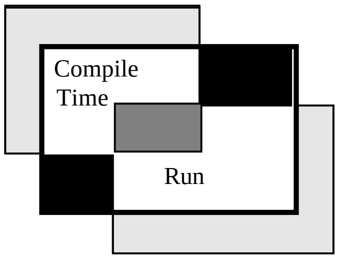
In the scenario of this diagram, checks are insufficient so obscure failures occur at run-time, varying from obscure run-time crashes to strangely wrong results to being lucky and getting away with it. Currently too much software development is based on programming until you are in the lucky state, known as *hacking*. This sorry situation in the industry must change by the adoption of better languages to remove the *ad hoc* nature of development.

Some feel that compiler checks are restrictive and that run-time checks are not efficient, so passionately defend this model, as programmers are supposedly trustworthy enough to remove the rest of the real consistencies. Although most programmers are conscientious and trustworthy people, this leaves too much to chance. You can produce defect-free software this way, as long as the programmer does not introduce the inconsistencies in the first place, but this becomes much more difficult as the size and complexity of a software system increases, and many programmers become involved. The real inconsistencies are often removed by hacking until the program works, with a resultant dependency on testing to find the errors in the first place. Sometimes companies depend on the customers to actually do the testing and provide feedback about the problems. While fault reporting is an essential path of communication from the customer, it must be regarded as the last and most costly line of defence.

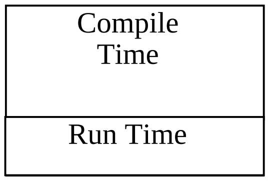
C and C++ are in this category. Software produced in these languages is prone to obscure failures.



The second figure, shows that the language detects inconsistencies beyond the real inconsistency box. These are false alarms. The run-time environment also doubles up on inconsistencies that the compiler has detected and removed, which results in run-time inefficiency. The language will be seen as restrictive, and the run-time as inefficient. You won't get any obscure crashes, but the language will get in the way of some useful computations. Pascal is often (somewhat unfairly) criticised for being too restrictive.



The above figure shows an even worse situation, where the compiler generates false alarms on fictional inconsistencies, does superfluous checks at run-time, but fails to detect real inconsistencies.



The best situation would be for a compiler to statically detect all inconsistencies without false alarms. However, it is not possible to statically detect all errors with the current state of technology, as a significant class of inconsistencies can only be detected at run-time; inconsistencies such as: divide by zero; array index out of bounds; and a class of type checks that are discussed in the section on RTTI and type casts.

The current ideal is to have the detectable and real inconsistency domains exactly coincide, with as few checks left to run-time as possible. This has two advantages: firstly, that your run-time environment will be a lot more likely to work without exceptions, so your software is safer; and secondly, that your software is more efficient, as you don't need so many run-time checks. A good language will correctly classify inconsistencies that can be detected at compile time, and those that must be left until run-time.

This analysis shows that as some inconsistencies can only be detected at run-time, and that such detection results in exceptions that exception handling is an exceedingly important part of software. Unfortunately, exception handling has not received serious enough attention in most programming languages.

Eiffel has been chosen for comparison in this critique as the language that is as close to the ideal as possible; that is, all inconsistencies are covered, while false alarms are minimised, and the detectable inconsistencies are correctly categorised as compile-time or run-time. Eiffel also pays serious attention to exception handling.

2.6 Types

In order to produce correct programs, syntax checks for conformance to a language grammar are not sufficient: we should also check semantics. Some semantics can be built into the language, but mostly this must be specified by the programmer about the system being developed.

Semantics checking is done by ensuring that a specification conforms to some schema. For example, the sentence: "The boy drank the computer and switched on the glass of water" is grammatically correct, but nonsense: it does not conform to the mental schema we have of computers and glasses of water. A programming language should include techniques for the detection of similar nonsense. The technique that enables detection of the above nonsense is types. We know from the computer's *type* that it does not have the property 'drinkable'. Types define an entity's properties and behaviour.

Programming languages can either be typed or untyped; typed languages can be statically typed or dynamically typed. Static typing ensures at compile time that only valid operations are applied to an entity. In dynamically typed languages, type inconsistencies are not detected until run-time. Smalltalk is a dynamically typed language, not an untyped language. Eiffel is statically typed.

C++ is statically typed, but there are many mechanisms that allow the programmer to render it effectively untyped, which means errors are not detected until a serious failure. Some argue that sometimes you might want to force someone to drink a computer, so without these facilities, the language is not flexible enough. The correct solution though is to modify the design, so that now the computer has the property drinkable. Undermining the type system is not needed, as the type system is where the flexibility should be, not in the ability to undermine the type system. Providing and modifying declarations is declarative programming. Eiffel tends to be declarative with a simple operational syntax, whereas C++ provides a plethora of operators.

Defining complex types is a central concept of object-oriented programming: "Perhaps the most important development [in programming languages] has been the introduction of features that support abstract data types (ADTs). These features allow programmers to add new types to languages that can be treated as though they were primitive types of the language. The programmer can define a type and a collection of constants, functions, and procedures on the type, while prohibiting any program using this type from gaining access to the implementation of the type. In particular, access to values of the type is available only through the provided constants, functions, and procedures." [Bruce 96].

Object-oriented programming also provides two specific ways to assemble new and complex types: "objects can be combined with other types in expressive and efficient ways (composition and hierarchy) to define new, more complex types." [Ege 96].

2.7 Redundancy and Checking

Redundant information is often needed to enable correctness checking. Type definitions define the elements in a system's universe, and the properties governing the valid combinations and interactions of the elements. Declarations define the entities in a system's universe. The compiler uses redundant information for consistency checking, and strips it away to produce efficient executable systems. Types are redundant information. You can program in an entirely typeless language: however, this would be to deny the progress that has been made in making programming a disciplined craft, that produces correct programs economically.

It is a misconception that consistency checks are 'training wheels' for student programmers, and that 'syntax' errors are a hindrance to professional programmers. Languages that exploit techniques of schema checking are often criticised as being restrictive and therefore unusable for real world software. This is nonsense and misunderstands the power of these languages. It is an immature conception; the best programmers realise that programming is difficult. As a whole, the computing profession is still learning to program.

While C++ is a step in this direction, it is hindered by its C base, importing such mechanisms as pointers with which you can undermine the logic of the type system. Java has abandoned these C mechanisms where they hinder: "The Java compiler employs stringent compile-time checking so that syntax-related errors can be detected early, before a program is deployed in service" [Sun 95]. The programming community has matured in the last few years, and while there was vehement argument against such checking in the past by those who saw it as restrictive and disciplinarian, the majority of the industry now accepts, and even demands it.

Checking has also been criticised from another point of view. This point of view says that checking cannot guarantee software quality, so why bother? The premise is correct, but the conclusion is wrong. Checking is neither necessary, nor sufficient to produce quality software. However, it is helpful and useful, and is a piece in a complicated jig-saw which should not be ignored.

In fact there are few things that are necessary for quality software production. Mainly, software quality is dependent on the skill and dedication of the people involved, not methodologies or techniques. There is nothing that is sufficient. As Fred Brooks has pointed out, there is no *Silver Bullet* [Brooks 95]. Good craftsmen choose the right tools and techniques, but the result is dependent on the skill used in applying the tools. Any tool is worthless in itself. But the *Silver Bullet* rationale is not a valid rationale against adopting better programming languages, tools and environments; unfortunately, Brooks' article has been misused.

Another example of consistency checking comes from the user interface world. Instead of correcting a user after an erroneous action, a good user interface will not offer the action as a possibility in the first place. It is cheaper to avoid error than to fix it. Most people drive their cars with this principle in mind: smash repair is time consuming and expensive.

Program development is a dynamic process; program descriptions are constantly modified during development. Modifications often lead to inconsistencies and error. Consistency checks help prevent such 'bugs', which can 'creep' into a previously working system. These checks help verify that as a program is modified, previous decisions and work are not invalidated.

It is interesting to consider how much checking could be integrated in an editor. The focus of many current generation editors is text. What happens if we change this focus from text to program components? Such editors might check not only syntax, but semantics. Signalling potential errors earlier and interactively will shorten development times, alerting programmers to problems, rather than wasting hours on changes which later have to be undone. Future languages should be defined very cleanly in order to enable such editor technology.

2.8 Encapsulation

There is much confusion about encapsulation, mostly arising from C++ equating encapsulation with *data hiding*. The Macquarie dictionary defines the verb *to encapsulate* as "*to enclose in or as in a capsule.*" The object-oriented meaning of encapsulation is to enclose related data, routines and definitions in a class capsule. This does not necessarily mean hiding.

Implementation hiding is an orthogonal concept which is possible because of encapsulation. Both data and routines in a class are classified according to their role in the class as interface or implementation.

To put this another way: first you encapsulate information and operations together in a class, then you decide what is visible, and what is hidden because it is implementation detail. Most often only the interface routines and data should appear at design time, the implementation details appearing later.

Encapsulation provides the means to separate the abstract interface of a class from its implementation: the interface is the visible surface of the capsule; the implementation is hidden in the capsule. The interface describes the essential characteristics of objects of the class which are visible to the exterior world. Like routines, data in a class can also be divided into characteristic interface data which should be visible, and implementation data which should be hidden. Interface data are any characteristics which might be of interest to the outside world. For example when buying a car, the purchaser might want to know data such as the engine capacity and horse-power, etc. However, the fact that it took John Engineer six days to design the engine block is of no interest.

Implementation hiding means that data can only be manipulated, that is updated, within the class, but it does not mean hiding interface data. If the data were hidden, you could never read it, in which case, classes would perform no useful function as you could only put data into them, but never get information out.

In order to provide implementation hiding in C++ you should access your data through C functions. This is known as data hiding in C++. It is not the data that is actually being hidden, but the access mechanism to the data. The access mechanism is the implementation detail that you are hiding. C++ has visible differences between the access mechanisms of constants, variables and functions. There is even a typographic convention of upper case constant names, which makes the differences between constants and variables visible. The fact that an item is implemented as a constant should also be hidden. Most non-C languages provide uniform functional access to constants, variables and value returning routines. In the case of variables, functional access means they can be read from the outside, but not updated. An important principle is that updates are centralised within the class.

Above I indicated that encapsulation was grouping operations and information together. Where do functions fit into this? The wrong answer is that functions are operations. Functions are actually part of the information, as a function returns information derived from an object's data to the outside world.

This theme and its adverse consequences, that place the burden of encapsulation on the programmer rather than being transparent, recur throughout this critique.

2.9 Safety and Courtesy Concerns

This critique makes two general types of criticism about 'safety' concerns and 'courtesy' concerns. These themes recur throughout this critique, as C and C++ have flaws that often compromise them. Safety concerns affect the *external* perception of the quality of the program; failure to meet them results in unfulfilled requirements, unsatisfied customers and program failures.

Courtesy concerns affect the *internal* view of the quality of a program in the development and maintenance process. Courtesy concerns are usually stylistic and syntactic, whereas safety concerns are semantic. The two often go together. It is a courtesy concern for an airline to keep its fleet clean and well maintained, which is also very much a safety concern.

Courtesy issues are even more important in the context of reusable software. Reusability depends on the clear communication of the purpose of a module. Courtesy is important to establish social interactions, such as communication. Courtesy implies inconvenience to the provider, but provides convenience to others. Courtesy issues include choosing meaningful identifiers, consistent layout and typography, meaningful and non-redundant commentary, etc. Courtesy issues are more than just a style consideration: a language design should directly support courtesy issues. A language, however, cannot enforce courtesy issues, and it is often pointed out that poor, discourteous programs can be written in any language. But this is no reason for being careless about the languages that we develop and choose for software development.

Programmers fulfilling courtesy and safety concerns provide a high quality service fulfilling their obligations by providing benefits to other programmers who must read, reuse and maintain the code; and by producing programs that delight the end-user.

The *programming by contract* model has been advocated in the last few years as a model for programming by which safety and courtesy concerns can be formally documented. Programming by contract documents the obligations of a client and the benefits to a provider in preconditions; and the benefits to the client and obligations of the provider in postconditions [Meyer 88], [Kilov and Ross 94].

2.10 Implementation and Deployment Concerns

Class implementors are concerned with the implementation of the class. Clients of the class only need to know as much information about the class as is documented in the abstract interface. The implementation is otherwise hidden.

Another aspect that is just as important to shield programmers from is deployment concerns. Deployment is how a system is installed on the underlying technology. If deployment issues are built into a program, then the program lacks portability, and flexibility. One kind of deployment concern is how a system is mapped to the available computing resources. For example, in a distributed system, this is what parts of the system are run in which location. As things can move around a distributed system, programmers should not build into their code location knowledge of other entities. Locations should be looked up in a directory.

Another deployment issue is how individual units of a system are plugged together to form an integrated whole. This is particularly important in OO, where several libraries can come from different vendors, but their combination results in conflicts. A solution to this is some kind of language that binds the units. Thus if you purchase two OO libraries, and they have clashes of any kind, you can resolve this deployment issue without having to change the libraries, which you might not be able to do anyway.

Programmers should not only be separated from implementation concerns of other units, but separated from deployment concerns as well.

2.11 Concluding Remarks

It is relevant to ask if grafting OO concepts onto a conventional language realises the full benefits of OO? The following parable seems apt: “No one sews a patch of unshrunk cloth on to an old garment; if he does, the patch tears away from it, the new from the old, and leaves a bigger hole. No one puts new wine into old wineskins; if he does, the wine will burst the skins, and then wine and skins are both lost. New wine goes into fresh skins.” *Mark 2:22*

We must abandon disorganised and error-prone practices, not adapt them to new contexts. How well can hybrid languages support the sophisticated requirements of modern software production? In my experience *bolt-on* approaches to object-orientation usually end in disaster, with the new tearing away from the old leaving a bigger hole.

Surely a basic premise of object-oriented programming is to enable the development of sophisticated systems through the adoption of the simplest techniques possible? Software development technologies and methodologies should not impede the production of such sophisticated systems.

3. C++ Specific Criticisms

3.1 Virtual Functions

This is the most complicated section in the critique, due to C++’s complex mechanisms. Although this issue is central as polymorphism is a key concept of OOP, feel free to skim if you want an overview, without the details.

In C++ the keyword `virtual` enables the possibility for a function to be polymorphic when it is overridden (redefined) in one or more descendant classes, but the `virtual` keyword is unnecessary, as any function which is redefined in a descendant class could be polymorphic. A compiler only needs to generate dynamic dispatch for truly polymorphic routines.

The problem in C++ is that if a parent class designer does not foresee that a descendant class might want to redefine a function, then the descendant class cannot make the function polymorphic. This is a most serious flaw in C++ because it reduces the flexibility of software components and therefore the ability to write reusable and extensible libraries.

C++ also allows functions to be overloaded, in which case the correct function to call depends on the arguments. The actual arguments in the function call must match the formal arguments of one of the overloaded functions. The difference between overloaded functions and polymorphic (overridden) functions is that with overloaded functions, the correct function to call is determined at compile-time; with polymorphic functions the correct function to call is determined at run-time.

When a parent class is designed the programmer can only guess that a descendant class might override or overload a function. A descendant class can overload a function at any time, but this is not the case for the more important mechanism of polymorphism, where the parent class programmer must specify that the routine is `virtual` in order for the compiler to set up a dispatch entry for the function in the class jump table. So the burden is on the programmer for something which could be automatically done by the compiler, and is done by the compiler in other languages. However, this is a relic from how C++ was originally implemented with Unix tools, rather than specialised compiler and linker support.

There are three options for overriding, corresponding to ‘must not’, ‘can’, and ‘must’ be overridden:

1) Overriding a routine is prohibited; descendant classes must use the routine as is.

2) A routine can be overridden. Descendant classes can use the routine as provided, or provide their own implementation as long as it conforms to the original interface definition and accomplishes at least as much.

3) A routine is abstract. No implementation is provided and each non-abstract descendent class must provide its own implementation.

The base class designer must decide options 1 and 3. Descendant class designers must decide option 2. A language should provide direct syntax for these options.

Option 1

C++ does not cater for the prohibition of overriding a routine in a descendant class. Even `private virtual` routines can be overridden. [Sakkinen 92] points out that a descendant class can redefine a `private virtual` function even though it cannot access the function in other ways.

Not using a virtual function is the closest, but in that case the routine can be completely replaced. This causes two problems. Firstly, a routine can be unintentionally replaced in a descendent. The redeclaration of a name within the same scope should cause a name clash; the compiler should report a ‘duplicate declaration’ syntax error as the entities inherited from the parent are included in the descendants namespace. Allowing two entities to have the same name within one scope causes ambiguity and other problems. (See the section on name overloading.)

The following example illustrates the second problem:

```
class A
{
   public:
   void nonvirt ();
   virtual void virt ();
}

class B : public A
{
   public:
   void nonvirt ();
   void virt ();
}

A a;
B b;
A *ap = &b;
B *bp = &b;

bp->nonvirt (); // calls B::nonvirt as
                // you would expect.
ap->nonvirt (); // calls A::nonvirt,
                // even though this
                // object is of type B.
ap->virt ();    // calls B::virt, the
                // correct version of
                // the routine for B
                // objects.
```

In this example, class B has extended or replaced routines in class A. `B::nonvirt` is the routine that should be called for objects of type B. It could be pointed out that C++ gives the client programmer flexibility to call either `A::nonvirt` or `B::nonvirt`, but this can be provided in a simpler more direct way: `A::nonvirt` and `B::nonvirt` should be given different names. That way the programmer calls the correct routine explicitly, not by an obscure and error prone trick of the language. The different name approach is as follows:

```
class B : public A
{
   public:
   void b_nonvirt ();
   void virt ();
}
B b;
B *bp = &b;
bp->nonvirt ();   // calls A::nonvirt
bp->b_nonvirt (); // calls B::b_nonvirt
```

Now the designer of class B has direct control over B’s interface. The application requires that clients of B can call both `A::nonvirt`, and `B::b_nonvirt`, which B’s designer has explicitly provided for. This is good object-oriented design, which provides strongly defined interfaces. C++ allows client programmers to play tricks with the class interfaces, external to the class, and B’s designer cannot prevent `A::nonvirt` from being called. Objects of class B have their own specialised `nonvirt`, but B's designer does not have control over B's interface to ensure that the correct version of `nonvirt` is called.

C++ also does not protect class B from other changes in the system. Suppose we need to write a class C that needs `nonvirt` to be `virtual`. Then `nonvirt` in A will be changed to `virtual`. But this breaks the `B::nonvirt` trick. The requirement of class C to have a `virtual` function forces a change in the base class, which affects all other descendants of the base class, instead of the specific new requirement being localised to the new class. This is against to the reason for OOP having loosely coupled classes, so that new requirements, and modifications will have localised effects, and not require changes elsewhere which can potentially break other existing parts of the system.

Another problem is that statements should consistently have the same semantics. The polymorphic interpretation of a statement like `a->f()` is that the most suitable implementation of `f()` is invoked for the object referred to by 'a', whether the object is of type `A`, or a descendent of `A`. In C++, however, the programmer must know whether the function `f()` is defined virtual or non-virtual in order to interpret exactly what `a->f()` means. Therefore, the statement `a->f()` is not implementation independent and the principle of implementation hiding is broken. A change in the declaration of `f()` changes the semantics of the invocation. Implementation independence means that a change in the implementation DOES NOT change the semantics, of executable statements.

If a change in the declaration changes the semantics, this should generate a compiler detected error. The programmer should make the statement semantically consistent with the changed declaration. This reflects the dynamic nature of software development, where you'll see perpetual change in program text.

For yet another case of the inconsistent semantics of the statement `a->f()` vs constructors, consult section 10.9c, p 232 of the C++ ARM. Neither Eiffel nor Java have these problems. Their mechanisms are clearer and simpler, and don't lead to the surprises of C++. In Java, everything is `virtual`, and to gain the effect where a method must not be overridden, the method may be defined with the qualifier `final`.

Eiffel allows the programmer to specify a routine as **frozen**, in which case the routine cannot be redefined in descendants.

Option 2

Using the function as is or overriding it should be left open for the programmers of descendant classes. In C++, the possibility must be enabled in the base class by specifying `virtual`. In object-oriented design, the decisions you decide not to make are as important as the decisions you make. Decisions should be made as late as possible. This strategy prevents mistakes being built into the system at early stages. By making early decisions, you are often stuck with assumptions that later prove to be incorrect; or the assumptions could be correct in one environment, but false in another, making software brittle and non-reusable.

C++ requires the parent class to specify potential polymorphism by virtual (although an intermediate class in the inheritance chain can introduce virtual). This prejudges that a routine might be redefined in descendants. This can be a problem because routines that aren't actually polymorphic are accessed via the slightly less efficient virtual table technique instead of a straight procedure call. (This is never a large overhead but object-oriented programs tend to use more and smaller routines making routine invocation a more significant overhead.) The policy in C++ should be that routines that might be redefined should be declared virtual. What is worse is that it says that non-virtual routines cannot be redefined, so the descendant class programmer has no control.

Rumbaugh et al put their criticism of C++'s virtual as follows: "C++ contains facilities for inheritance and run-time method resolution, but a C++ data structure is not automatically object-oriented. Method resolution and the ability to override an operation in a subclass are only available if the operation is declared virtual in the superclass. Thus, the need to override a method must be anticipated and written into the origin class definition. Unfortunately, the writer of a class may not expect the need to define specialised subclasses or may not know what operations will have to be redefined by a subclass. This means that the superclass often must be modified when a subclass is defined and places a serious restriction on the ability to reuse library classes by creating sub-classes, especially if the source code library is not available. (Of course, you could declare all operations as virtual, at a slight cost in memory and function-calling overhead.)" [RBPEL91]

Virtual, however, is the wrong mechanism for the programmer to deal with. A compiler can detect polymorphism, and generate the underlying virtual code, where and only where necessary. Having to specify virtual burdens the programmer with another bookkeeping task. This is the main reason why C++ is a weak object-oriented language as the programmer must constantly be concerned with low level details, which should be automatically handled by the compiler.

Another problem in C++ is mistaken overriding. The base class routine can be overridden unwittingly. The compiler should report an erroneous name redefinition within the same name space unless the descendant class programmer specifies that the routine redefinition is really intended. The same name can be used, but the programmer must be conscious of this, and state this explicitly, especially in environments where systems are assembled out of preexisting components. Unless the programmer explicitly overrides the original name a syntax error should report that the name is a duplicate declaration. C++, however, adopted the original approach of Simula. This approach has been improved upon, and other languages have adopted better, more explicit approaches, that avoid the error of mistaken redefinition.

The solution is that `virtual` should not be specified in the parent. Where run-time polymorphic dynamic-binding is required, the child class should specify `override` on the function. When compile-time static-binding is required, the child class should specify `overload` on the function. This has the advantages: in the case of polymorphic functions, the compiler can check that the function signatures conform; and in the case of overloaded functions that the function signatures are different in some respect. The second advantage would be that during the maintenance phases of a program, the original programmer's intention is clear. As it is, later programmers must guess if the original programmer had made some kind of error in choosing a duplicate name, or whether overloading was intended.

In Java, there is no `virtual` keyword; all methods are potentially polymorphic. Java uses direct call instead of dynamic method lookup when the method is `static`, `private` or `final`. This means that there will be non-polymorphic routines that must be called dynamically, but the dynamic nature of Java means further optimisation is not possible.

Eiffel and Object Pascal cater for this option as the descendant class programmer must specify that redefinition is intended. This has the extra benefit that a later reader or maintainer of the class can easily identify the routines that have been redefined, and that this definition is related to a definition in an ancestor class without having to refer to ancestor class definitions. Thus option 2 is exactly where it should be, in descendant classes.

Both Eiffel and Object Pascal optimise calls: they only generate dispatch table entries for dynamic binding where a routine is truly polymorphic. How this is possible is covered in the section on global analysis.

Option 3

The `pure virtual` function caters for leaving a function abstract, that is a descendent class must provide its implementation if it is to be instantiated. Any descendants that do not define the routine are also abstract classes. This concept is correct, but see the section on `pure virtual` functions for criticism of the terminology and syntax.

Java also has abstract methods, and in Eiffel, the implementation is marked as **deferred**.

Summary

The main problem with `virtual` is that it forces the base class designer to guess that a function might be polymorphic in one or more derived classes. If this requirement is not foreseen, or not included as an optimisation to avoid dynamically dispatched calls, the possibility is effectively closed, rather than being left open. As implemented in C++, virtual coupled with the independent notion of overloading make an error prone combination.

`Virtual` is a difficult notion to grasp. The related concepts of polymorphism and dynamic binding, redefinition, and overriding are easier to grasp, being oriented towards the problem domain. Virtual routines are an implementation mechanism which instruct the compiler to set up entries in the class's virtual table; where global analysis is not done by the compiler, leaving this burden to the programmer. Polymorphism is the 'what', and virtual is the 'how'. Smalltalk, Objective-C, Java, and Eiffel all use a different mechanism to implement polymorphism.

Virtual is an example of where C++ obscures the concepts of OOP. The programmer has to come to terms with low level concepts, rather than the higher level object-oriented concepts. Virtual leaves optimisation to the programmer. Other approaches leave the optimisation of dynamic dispatch to the compiler, which can remove 100% of cases where dynamic dispatch is not required. Interesting as underlying mechanisms might be for the theoretician or compiler implementor, the practitioner should not be required to understand or use them to make sense of the higher level concepts. Having to use them in practice is tedious and error-prone, and can prevent the adaptation of software to further advances in the underlying technology and execution mechanisms (see concurrent programming), and reduces the flexibility and reusability of the software.

3.2 Global Analysis

[P&S 94] note that there are two *world assumptions* about type safety. The first is the *closed-world* assumption, where all parts of the program are known at compilation time, and type checking is done for the entire program. The second is the *open-world* assumption, where type checking is done independently for each module. The open-world assumption is useful when developing and prototyping. However, "When a finished product has matured, it makes sense to adopt the closed-world assumption, since it enables more advanced compilation techniques. Only when the entire program is known, is it possible to perform global register allocation, flow analysis, or dead code detection." [P&S 94].

One of the major problems with C++ is the way analysis is divided between the compiler, which works under the open-world assumption, and the linker which is depended on to do very limited closed-world analysis. Closed-world or *global* analysis is essential for two reasons: firstly, to ensure that the assembled system is consistent; and secondly to remove burden from the programmer by providing automatic optimisations.

The main burden that can be removed from the programmer is that of a base class designer having to help the compiler build class virtual tables with the virtual function modifier. As explained in the section on virtual functions, this adversely effects software flexibility. Virtual tables should not be built when a class is compiled: rather virtual tables should only be built when the entire system is assembled. During the system assembly (linker) phase, the compiler and linker can entirely determine which functions need virtual table entries. Other burdens are that the programmer must use operators to help the compiler with information in other modules it cannot see, and the maintenance of header files.

In Eiffel and Object Pascal, global analysis of the entire system is done to determine the truly polymorphic calls and accordingly construct the virtual tables. In Eiffel this is done by the compiler. In Object Pascal, Apple extended the linker to perform global analysis. Such global analysis is difficult in a C/Unix style environment, so in C++ it was not included, leaving this burden to the programmer.

In order to remove this burden from the programmer, global analysis should have been put in the linker. However, as C++ was originally implemented as the Cfront preprocessor, necessary changes to the linker weren't undertaken. The early implementations of C++ were a patchwork, and this has resulted in many holes. The design of C++ was severely limited by its implementation technology, rather than being guided by the principles of better language design, which would require dedicated compilers and linkers. That is, C++ has been severely limited by its original experimental implementation.

I am now convinced that such technology dependence has severely damaged C++ as an object-oriented language and as a high level language. A high level language removes the bookkeeping burden from the programmer and places them in the compiler, which is the primary aim of high level languages. Lack of global or closed-world analysis is a major deficiency of C++, which leaves C++ substantially lacking when compared to languages such as Eiffel. As Eiffel insists on *system level validity* and therefore global analysis, it means that Eiffel implementations are more ambitious than C++ implementations, and this is a major reason why Eiffel implementations have been slower to appear.

Java dynamically loads pieces of software and links them into a running system as required. Thus static compile-time global analysis is not possible, as Java is designed to be dynamic. However, Java has made the valid assumption that all methods are virtual. This is one reason why Java and Eiffel are substantially different tools, although Eiffel has recently introduced *Dynamic Linking in Eiffel* (DLE).

3.3 Type-safe linkage

The C++ ARM explains that type-safe linkage is not 100% type safe. If it is not 100% type-safe, then it is unsafe. Statistical analysis showed that in the Challenger disaster, the probability against an individual O-ring failure was .997. But in a combination of 6 this small margin for failure became significant, meaning the combination was very likely to fail. In software, we often find strange combinations cause failure. It is the primary objective of OO to reduce these strange combinations.

It is the subtle errors that cause the most problems, not the simple or obvious ones. Often such errors remain undetected in the system until critical moments. The seriousness of this situation cannot be underestimated. Many forms of transport, such as planes, and space programs depend on software to provide safety in their operation. The financial survival of organisations can also depend on software. To accept such unsafe situations is at best irresponsible.

C++ type safe linkage is a huge improvement over C, where the linker will link a function f (p1, ...) with parameters to any function f (), maybe one with no or different parameters. This results in failure at run time. However, since C++ type safe linkage is a linker trick, it does not deal with all inconsistencies like this.

The C++ ARM summarises the situation as follows - “Handling all inconsistencies - thus making a C++ implementation 100% type-safe - would require either linker support or a mechanism (an environment) allowing the compiler access to information from separate compilations.”

So why do C++ compilers (at least AT&T's) not provide for accessing information from separate compilations? Why is there not a specialised linker for C++, that actually provides 100% type safety? C++ lacks the global analysis of the previous section. Building systems out of preexisting elements is the common Unix style of software production. This implements a form of reusability, but not in the truly flexible and consistent manner of object-oriented reusability.

In the future, Unix might be replaced by object-oriented operating systems, that are indeed ‘open’ to be tailored to best suit the purpose at hand. By the use of pipes and flags, Unix software elements can be reused to provide functionality that approximates what is desired. This approach is valid and works with efficacy in some instances, like small in-house applications, or perhaps for research prototyping, but is unacceptable for widespread and expensive software, or safety critical applications. In the last ten years the advantages of integrated software have been acknowledged. Classic Unix systems don't provide those advantages. Integrated systems are more ambitious, and place more demands on their developers, but this is the sort of software now being demanded by end users. Systems that are cobbled together are unacceptable. Today the emphasis is on *software component technologies* such as the public domain *OpenDoc* or Microsoft's *OLE*.

A further problem with linking is that different compilation and linking systems should use different name encoding schemes. This problem is related to type-safe linkage, but is covered in the section on 'reusability and compatibility'.

Java uses a different dynamic linking mechanism, which is well defined and does not use the Unix linker. Eiffel does not depend on the Unix or other platform linkers to detect such problems. The compiler must detect these problems.

Eiffel defines **system-level validity**. An Eiffel compiler is therefore required to perform closed-world analysis, and not rely on linker tricks. You can thus be sure that Eiffel programs are 100% type safe. A disadvantage of Eiffel is that compilers have a lot of work to do. (The common terminology is 'slow', but that is inaccurate.) This is overcome to some extent by Eiffel's melting-ice technology, where changes can be made to a system, and tested without the need to recompile every time.

To summarise the last two sections: global or closed-world analysis is needed for two reasons: consistency checks and optimisations. This removes many burdens from the programmer, and its lack is a great shortcoming of C++.

3.4 Function Overloading

C++ allows functions to be overloaded if the arguments in the signature are different types. Overloaded functions are different to polymorphic functions: for each invocation the correct function is selected at compile time; with polymorphic functions, the correct function is bound dynamically at run-time. Polymorphism is achieved by redefining or overriding routines. Be careful not to confuse overriding and overloading. Overloading arises when two or more functions share a name. These are disambiguated by the number and types of the arguments. Overloading is different to multiple dispatching in CLOS, as multiple dispatching on argument types is done dynamically at run-time.

[Reade 89] points out the difference between overloading and polymorphism. Overloading means the use of the same name in the same context for different entities with completely different definitions and types. Polymorphism though has one definition, and all types are subtypes of a principle type. C. Strachey referred to polymorphism as parametric polymorphism and overloading as ad hoc polymorphism. The qualification mechanism for overloaded functions is the function signature.

Overloading can be useful as these examples show:

```
max (int, int);
max (real, real);
```

This will ensure that the best max routine for the types `int` and `real` will be invoked. Object-oriented programming, however, provides a variant on this. Since the object is passed to the routine as a hidden parameter ('this' in C++), an equivalent but more restricted form is already implicitly included in object-oriented concepts. A simple example such as the above would be expressed as:

```
int i, j;
real r, s;
i.max (j);
r.max (s);
```

but i.max (r) and r.max (j) result in compilation errors because the types of the arguments do not agree. By operator overloading of course, these can be better expressed, i max j and r max s, but min and max are peculiar functions that could accept two or more parameters of the same type so they can be applied to a arbitrarily sized list. So the most general code in Eiffel style syntax will be something like:

il: *COMPARABLE_LIST* [*INTEGER*]
r*l*: *COMPARABLE_LIST* [*REAL*]

i := *il.max*
r := r*l.max*

The above examples show that the object-oriented paradigm, particularly with genericity can achieve function overloading, without the need for the function overloading of C++. C++, however, does make the notion more general. The advantage is that more than one parameter can overload a function, not just the implicit current object parameter.

Another factor to consider is that overloading is resolved at compile time, but overriding at run-time, so it looks as if overloading has a performance advantage. However, global analysis can determine whether the *min* and *max* functions are at the end of the inheritance line, and therefore can call them directly. That is, the compiler examines the objects *i* and *r*, looks at their corresponding *max* function, sees that at that point no polymorphism is involved, and so generates a direct call to *max*. By contrast, if the object was *n* which was defined to be a *NUMBER* which provided the abstract *max* function from which *REAL.max* and *INTEGER.max* were derived, then the compiler would need to generate a dynamically bound call, as *n* could refer to either a *INTEGER* or a *REAL*.

If it is felt that C++'s scheme of having parameters of different types is useful, it should be realised that object-oriented programming provides this in a more restricted and disciplined form. This is done by specifying that the parameter needs to conform to a base class. Any parameter passed to the routine can only be a type of the base class, or a subclass of the base class. For example:

```
A.f (B someB) {...};
class B ...;
class D : public B ...
A a;
```

```
D d;
a.f (d);
```

The entity ‘d’ must conform to the class ‘B’, and the compiler checks this.

The alternative to function overloading by signature, is to require functions with different signatures to have different names. Names should be the basis of distinction of entities. The compiler can cross check that the parameters supplied are correct for the given routine name. This also results in better self-documented software. It is often difficult to choose appropriate names for entities, but it is well worth the effort.

[Wiener 95] contributes a nice example on the hazards of virtual functions with overloading:

```
class Parent
{
   public:
      virtual int doIt (int v)
      {
         return v * v;
      }
};

class Child : public Parent
{
   public:
      int doIt (int v,
                int av = 20)
      {
         return v * av;
      }
};

void main()
{
   int i;
   Parent *p = new Child();
   i = p->doIt(3);
}
```

What is the value in `i` after execution of this program? One might expect 60, but it is 9 as the signature of `doIt` in `Child` does not match the signature in `Parent`. It therefore does not override the `Parent doIt`, merely overloads it, and the default is unusable.

Java also provides *method overloading*, where several methods can have the same name, but have different signatures.

The Eiffel philosophy is not to introduce a new technique, but to use genericity, inheritance and redefinition. Eiffel provides covariant signatures, which means the signatures of descendant routines do not have to match exactly, but they do have to conform, according to Eiffel’s strong typing scheme.

Eiffel uses covariance with anchored types to implement examples such as max. The Vintage 95 Kernel Library specifies max as:

max (*other*: **like** *Current*): **like** *Current*

This says that the type of the argument to max must conform to the type of the current class. Therefore you get the same effect by redefinition without the overloading concept. You also get type checking to see that the parameter conforms to the current object. Genericity is also a mechanism that overcomes most of the need for overloading.

3.5 The Nature of Inheritance

Inheritance is a close relationship providing a fundamental OO way to assemble software components, along with composition and genericity. Objects that are instances of a class are also instances of all ancestors of that class. For effective object-oriented design the consistency of this relationship should be preserved. Each redefinition in a subclass should be checked for consistency with the original definition in an ancestor class. A subclass should preserve the requirements of an ancestor class. Requirements that cannot be preserved indicate a design error and perhaps inheritance is not appropriate. Consistency due to inheritance is fundamental to object-oriented design. C++’s implementation of non-virtual overloading, means that the compiler does not check for this consistency. C++ does not provide this aspect of object-oriented design.

Inheritance has been classified as ‘syntactic’ inheritance and ‘semantic’ inheritance. Saake et al describe these as follows: “Syntactic inheritance denotes inheritance of structure or method definitions and is therefore related to the reuse of code (and to overriding of code for inherited methods). Semantic inheritance denotes inheritance of object semantics, ie of objects themselves. This kind of inheritance is known from semantic data models, where it is used to model one object that appears in several roles in an application.” [SJE 91]. Saake et al concentrate on the semantic form of inheritance. Behavioural or semantic inheritance expresses the role of an object within a system.

Wegner, however, believes code inheritance to be of more practical value. He classifies the difference between syntactic and semantic inheritance as code and behaviour hierarchies [Weg 91] (p43). He suggests these are rarely compatible with each other and are often negatively correlated. Wegner also poses the question of “How should modification of inherited attributes be constrained?” Code inheritance provides a basis for modularisation. Behavioural inheritance provides modelling by the ‘is-a’ relationship. Both are useful in their place. Both require consistency checks that combinations due to inheritance actually make sense.

It seems that inheritance is most powerful in the most restrictive form of a semantics preserving relationship; a subclass should preserve the assumptions of ancestor classes.

Meyer [Meyer 96a and 96b] has also produced a classification of inheritance techniques. In his *taxonomy* he identifies 12 uses of inheritance, all of which he finds useful. This analysis also gives a good idea of when inheritance can be used, and when it should not.

Software components are like jig-saw pieces. When assembling a jig-saw the shape of the pieces must fit, but more importantly, the resulting picture must make sense. Assembling software components is more difficult. A jig-saw is reassembling a picture that was complete before. Assembling software components is building a picture that has never been seen before. What is worse, is that often the jig-saw pieces are made by different programmers, so when the whole system is assembled, the pictures must fit.

Inheritance in C++ is like a jig-saw where the pieces fit together, but the compiler has no way of checking that the resultant picture makes sense. In other words C++ has provided the syntax for classes and inheritance but not the semantics. Reusable C++ libraries have been slow to appear, which suggests that C++ might not support reusability as well as possible. By contrast Java, Eiffel and Object Pascal are packaged with libraries. Object Pascal went very much in hand with the MacApp application framework. Java has been released coupled with the Java API, a comprehensive library. Eiffel is also integrated with an extremely comprehensive library, which is even larger than Java's. In fact the concept of the library preceded Eiffel as a project to reclassify and produce a taxonomy of all common structures used in computer science. [Meyer 94].

3.6 Multiple Inheritance

Both Eiffel and C++ provide multiple inheritance. Java does not, claiming it results in many problems. Instead Java provides *interfaces*, which are similar to Objective C's protocols. Sun claims interfaces provide all the desirable features of multiple inheritance.

Sun's claim that multiple inheritance results in problems is true particularly in the way that C++ has implemented multiple inheritance. What seems like a simple generalisation of inheriting from multiple classes instead of just one, turns out to be non-trivial. For example, what should be the policy if you inherit an item of the same name from two classes? Are they compatible? If so should they be merged into a single entity? If not, how do you disambiguate them? And so the list goes on.

Java's interface mechanism implements multiple inheritance, with one important difference: the inherited interfaces must be abstract. This does obviate the need to choose between different implementations, as with interfaces there are no implementations. Java allows the declaration of constant fields in an interface. Where these are multiply inherited, they merge to form one entity so that no ambiguity arises, but what happens if the constants have different values?

Since Java does not have multiple inheritance, you cannot do *mixins* as you can in C++ and Eiffel. Mixin is the ability to inherit sets of non-abstract routines from different classes to build a new complex class. For example, you might want to import utility routines from a number of different sources. However, you can achieve the same effect using composition instead of inheritance, so this is probably not a great minus against Java.

Eiffel solves multiple inheritance problems without having to introduce a separate, interface mechanism.

Some feel that single inheritance is elegant by itself, but that multiple inheritance is not. This is one particular standpoint.

BETA [Madsen 93] falls into the 'multiple inheritance is inelegant' category: "Beta does not have multiple inheritance, due to the lack of a profound theoretical understanding, and also because the current proposals seem technically very complicated." They cite Flavors as a language that mixes classes together, where according to Madsen, the order of inheritance matters, that is inheriting (A, B) is different from inheriting (B, A).

Ada 95 is also a language that avoids multiple inheritance. Ada 95 supports single inheritance as the *tagged type extension*.

Others feel that multiple inheritance can provide elegant solutions to particular modelling problems so is worth the effort. Although, the above list of questions arising from multiple inheritance is not complete, it shows that the problems with multiple inheritance can be systematically identified, and once the problems are recognised, they can be solved elegantly. While [Sakkinen 92] goes into the problems of multiple inheritance in great depth, he defends it.

Eiffel has taken the approach that multiple inheritance poses some interesting and challenging problems, but rises to the challenge, and solves them elegantly. Nor does the order of inheritance matter. All resolutions that the programmer must specify are given in the inheritance clause of a class. This includes *renaming* to ensure that multiple features inherited with the same name end up as multiple features with unambiguous names, *redefining*, new *export* policies for inherited features, *undefining*, and disambiguating with *select*. In all cases, the action taken by the compiler, whether using fork or join semantics is made clear, and the programmer has complete control.

C++ has a different disambiguation mechanism to Eiffel. In Eiffel, one or both of the features must be given a different name in the renames clause. In C++ the members must be disambiguated using the *scope resolution operator* '::'. The advantage of the Eiffel approach is that the ambiguity is dealt with declaratively in one place. Eiffel's inheritance clause is considerably more complex than C++'s, but the code is considerably simpler, more robust and flexible, which is the advantage of the declarative approach as against the operator approach. In C++, you must use the scope resolution operator in the code, every time you run into an ambiguity problem between two or more members. This clutters the code, and makes it less malleable, as if anything changes that affects the ambiguity, you potentially have to change the code everywhere, where the ambiguity occurs.

According to [Stroustrup 94] section 12.8, the ANSI committee considered renaming, but the suggestion was blocked by one member who insisted that the rest of the committee go away and think about it for two weeks. The example in section 12.8 shows how the effect of renaming is achieved, without explicit renaming. The problem is, if it took this group of experts two weeks to work this out, what chance is there for the rest of us?

The scope resolution operator is used for more than just multiple inheritance disambiguation. Since ambiguities could be avoided by cleaner language design, the scope resolution operator is an ugly complication.

The question of whether the order of declaration of multiple parents matters in C++ is complex. It does affect the order in which constructors are called, and can cause problems if the programmer does really want to get low level. However, this would be considered poor programming practice.

Another difference between C++ and Eiffel is direct repeated inheritance. Eiffel allows:

```
class B inherit A, A end
```

but

```
class B : public A, public A {  };
```

is disallowed in C++.

3.7 Virtual Classes

The meaning of the keyword `virtual` is quite different when used in the context of a class to the context of a function: with a class it means that multiply inherited features are merged; with a function it means polymorphism. Virtual class does not mean that members in the class are all polymorphic. In fact the two uses of virtual actually mean quite the opposite of each other: virtual functions mean that there could be more than one function; virtual classes mean that if the class is multiply inherited, you only get a single copy.

C++ saves on keywords by overloading one keyword in several contexts, even though the uses have different or even opposite meanings. Static is another case, which is used in three different contexts. The keyword count metric does not show that C++ is a small non-complex language: less keywords have made C++ more complex and confusing.

So what do virtual classes do? If class D multiply inherits class A via classes B and C, then if D wants to inherit only a single shared copy of A, the inheritance of A must be specified as `virtual` in both B and C. C++ virtual classes raise two questions. Firstly, what happens if A is declared virtual in only one of B or C? Secondly, what if another class E wants to inherit multiple copies of A via B and C? In C++, the virtual class decision must be made early, reducing the flexibility that might be required in the assembly of derived classes. In a shared software environment different vendors might supply classes B and C. It should be left to the implementor of class D or E, exactly how to resolve this problem. And this is the simplest case: what if A is inherited via more than two paths, with more than two levels of inheritance? Flexibility is key to reusable software. You cannot envisage when designing a base class all the possible uses in derived classes, and attempting to do so considerably complicates design.

As Java has no multiple inheritance, there is no problem to be solved here.

The Eiffel mechanism allows two classes D and E inheriting multiple copies of A to inherit A in the appropriate way independently. You do not have to choose in intermediate classes whether A is virtual, ie., inherited as a single copy, or not. The inheritance is more flexible and done on a feature by feature basis, and each feature from A will either fork, in which it becomes two new features; or join, in which case there is only one resultant feature. The programmer of each descendant class can decide whether it is appropriate to fork or join each feature independently of the other descendants, or any policy in A.

The fine grained approach of Eiffel is a significant benefit over C++. While the Eiffel approach is more sophisticated and flexible, the syntax is far simpler, and the concepts are easier to understand.

3.8 Templates

Templates are C++'s mechanism to implement the concept of *genericity*. Templates are much the same as *parameterised classes*, which is the mechanism Eiffel uses for genericity. Genericity is a major feature of Ada and Algol 68 and is a valuable addition to C++. Some see genericity as a more fundamental software assembly mechanism than inheritance, and certainly less problematic. Ada is an example where genericity is more fundamental than inheritance. In C++'s Standard Template Library (STL), genericity is used almost exclusively instead of inheritance. Meyer [Meyer 88] states that genericity is an essential part of an object-oriented language. [P&S 94] see genericity as a mechanism that achieves type substitution, which you cannot do with inheritance. Thus genericity is essential as a complementary concept to inheritance.

Genericity allows you to build collections of items, where the type of items is known, and items can be retrieved from the collection as that type, without type casting. In a language without genericity you code a *LIST* class, and objects of any type can be added to lists. If the list is only for shopping items, it makes semantic nonsense to add a person to the list. Without genericity there is no static type check to ensure you can't add people to your shopping list. You might be able to catch this occurrence at run time, but the advantage of static typing is lost.

Without genericity you could code specific lists for shopping items, people, and every other item you could put in lists. The basic functionality of all lists is the same, but you must duplicate effort, and manually replicate code. That is you must duplicate effort if you are going to preserve semantics and be type safe.

Languages such as Eiffel and C++ allow you to declare a *LIST* of *shopping items*, so the compiler can ensure that you cannot add people to such a list. You can also easily add lists that contain any other type of entity, just by a simple declaration. You do not have to manually replicate the basic functionality of the list for every type of element you are going to put in it.

This has lead to a criticism of the C++ template mechanism that you get 'code bloat'. That is for every type based on a template definition the compiler might replicate the code. Seeing that the purpose of templates is to save the programmer from manual replication, this does not seem like a bad thing. A good implementation of C++ will avoid 'code bloat' where possible. In fact it is allowed for in the C++ ARM: "This can cause the generation of unnecessarily many function definitions. A good implementation might take advantage of the similarity of such functions to suppress spurious replications."

Thus I don't criticise C++ as others have done on the basis of 'code bloat'. The whole concept of generics and templates is simple and yet powerful, and allows the generation of quite sophisticated programs from simple specifications. If you are overly worried about 'code bloat', simply do not use genericity. As [Stroustrup 94] points out "What you don't use, you don't pay for." This is a good principle for compiler implementors. Many people will use genericity though, as few will find it practical to code a different kind of *LIST* for every possible list element.

While the concept of genericity and templates is correct, there are several problems with templates in C++. The syntax leaves a lot to be desired. Readers can of course form their own opinions of that. However, again C++ masks what is a simple and powerful mechanism with complicated syntax, so people will baulk at using it. There are examples of where the quirky syntax is a trap for young players [Stroustrup 94]. For example, declaring a list of a list of integers would easily be notated:

```
List<List<int>> a;
```

However, this results in a syntax error as '>>' is the right shift or output operator. You must notate this as '> >':

```
List<List<int> > a;
```

Further, "template" is confusing terminology, as the conceptual view is that a class is a template for a set of objects. "Object-oriented languages allow one to describe a template, if you will, for an entire set of objects. Such a template is called a class." [Ege 96]. This is not the meaning of the C++ term template, which refers to genericity.

Another more serious problem is that there is no constraint on the types that can be used as the parameters to the templates; the coder of a template class can make no assumptions about the type of the generic parameter. Thus the class coder cannot issue a function call from within the template class to the generic type without a type cast.

As the ARM says on this topic: "Specifying no restrictions on what types can match a type argument gives the programmer the maximum flexibility. The cost is that errors - such as attempting to sort objects of a type that does not have comparison operators - will not in general be detected until link time."

This shows the need for at least an optional type constraint on the actual types passed to the template. Eiffel has such optional constraints in the form of *constrained genericity*. For example:

```
class SORTED_LIST [T -> COMPARABLE]
    ...
    feature
        insert (item: T) is ... end
end
```

ensures that the type of the item to insert has appropriate comparison operators from type *COMPARABLE* in order to insert item in the right place in the *SORTED_LIST*. Note that multiple inheritance is important, so that any type eligible for insertion in the *SORTED_LIST* includes the comparison operators.

Java, alas has no genericity mechanism. The Java recommendation is to use type casts when ever retrieving an object from a container class [Flan 96].

[P&S 94] have a good chapter on genericity. Genericity is the ability to build a derived class from a base class by type substitution. Compare this with inheritance, where you can add class members and redefine inherited routines. They criticise the parameterised class/template mechanisms of Eiffel and C++ for three reasons: firstly, there are two kinds of class, generic and non-generic; secondly, you can apply generic instantiation only once; and thirdly, a generic instance is not a subclass.

BETA uses a different mechanism, *virtual binding*, which is more flexible than the Eiffel/C++ parameterised classes, but [P&S 94] shows that you can produce derived classes that are not statically type correct.

A significant problem with the parameterised class mechanism is that the base class designer must think about it in advance, and then only the types nominated in the parameter list can be substituted. This reduces flexibility. [P&S 94] suggests a genericity mechanism known as *class substitution*, which make inheritance and genericity orthogonal rather than independent concepts. Class substitution has the advantage that a base class designer does not need to design genericity into the base class, any subclass can perform class substitution; and any type in the base class may be substituted, not only those given in the parameter list. Furthermore, class substitution can be applied repeatedly, whereas instantiation of a parameterised class can be done only once.

An example of class substitution in Eiffel like syntax is:

```
class A
    feature
        x, y: T

        assign is
            do
                x := y
            end
end
```

This can be modified using class substitution:

```
A [T <- INTEGER]
A [T <- ANIMAL]
```

You can also use constrained genericity with exactly the same syntax that Eiffel now has, as in the *SORTED_LIST* example, except that semantically the [*T* -> *COMPARABLE*] only specifies that any class substituting *T* must be a subclass of *COMPARABLE*. [*T* -> *COMPARABLE*] is not a parameter list though. You can build new types out of sorted list:

```
SORTED_LIST [T <- INTEGER]
SORTED_LIST [T <- STRING]
```

Java might be in the best position to implement this flexible class substitution mechanism for genericity, as it has not implemented genericity yet. Eiffel and C++ could extend their mechanisms, but then there would be two ways of doing the same thing, except the class substitution mechanism is more flexible than parameterised classes. I do not know of any languages that implement class substitution as yet, and other consequences must be thought through before adding it to languages, so don't dispose of your Eiffel and C++ compilers just yet!

3.9 Name Overloading

Clear names are fundamental in producing self-documenting software helping to produce maintain-able and reusable software components. Names are fundamental in freeing programmers from low level manipulation of addresses. Naming is the basis for differentiating between different entities in a software module. In programming, when we use the term name, we usually mean identifier. To be precise, a name is a label which can refer to more than one entity, in which case the name is ambiguous. An identifier is a name that unambiguously identifies an entity. (To be mathematical, a name is a relation, an identifier is a function.) Where a name is ambiguous, it needs qualification to form an identifier to the entity. For example, there could be two people named John Doe; to disambiguate the reference, you would qualify each as John Doe *of Washington* or John Doe *of New York*.

Name overloading allows the same name to refer to two or more different entities. The problem with an ambiguous name is whether the resultant ambiguity is useful, and how to resolve it, as ambiguity weakens the usefulness of names to distinguish entities.

Name overloading is useful for two purposes. Firstly, it allows programmers to work on two or more modules without concern about name clashes. The ambiguity can be tolerated as within the context of each module the name unambiguously refers to a unique entity; the name is qualified by its surrounding environment. Secondly, name overloading provides polymorphism, where the same name applied to different types refers to different implementations for those types. Polymorphism allows one word to describe 'what' is computed. Different classes might have different implementations of 'how' a computation is done. For example 'draw' is an operation that is applicable to all different shapes, even though circles and squares, etc., are 'drawn' differently.

These two uses of name overloading provide a powerful concept. The use of the same name in the same context must be resolved. Errors can result from ambiguity, in which case the programmer must differentiate between entities with some form of qualification of the name. A common way to do this is to introduce extra distinguishing names. For example, in a group of people where two or more share the same first name, they can be distinguished by their surname. Similarly a unique first name will distinguish the members of a family with a common surname.

This is analogous to classes, where each class in a system is given a unique name. Each member within a class is also given a unique name. Where two objects with members of the same name are used within the same context, the object name can qualify the members. In this case the dot operator acts as a qualifier, for example, a.mem and b.mem.

Locals in a recursive environment are an example of ambiguity which is resolved at run-time. A single local identifier in the static text of a function can refer to many entities. When the function is called recursively, the name is qualified by the call history of the function to give the exact memory cell where it resides.

Many block structured languages provide overloading by scoping. Scoping allows the same name to be used in different contexts without clash or confusion, but nested blocks have a subtle problem. Names in an outer block are in scope in inner blocks, but many languages allow a name to be overloaded in an inner block, creating a 'scope hole' hiding the outer entity, preventing it from being accessed. The name in the inner block has no relationship with the entity of the same name in the outer block. Textually nested blocks 'inherit' named entities from outer blocks. Inheritance accomplishes this in object-oriented languages, eliminates the need to textually nest entities, and accomplishes textual loose coupling. Nesting results in tightly coupled text.

Contrary to most languages, a name should not be overloaded while it is in scope. The following example illustrates why:

```
{
  int i;
  {
    int i;  // hide the outer i.
    i = 13; // assign to the inner i.
    // Can't get to the outer i here.
    // It is in scope, but hidden.
  }
}
```

Now delete the inner declaration:

```
{
   int i;
   {
      i = 13; // Syntactically valid,
              // but not the intention.
   }
}
```

The inner overloaded declaration is removed, and references to that name do not result in syntax errors due to the same name being in the outer environment. The inner instruction now mistakenly changes the value of the outer entity. A compiler cannot detect this situation unless the language definition forbids nested redeclarations. E.W. Dijkstra uses similar reasoning in 'An essay on the Notion: "The Scope of Variables"' in "A Discipline of Programming," [Dijkstra 76].

The above example demonstrates how nesting results in less maintainable programs due to tight coupling between the inner and outer blocks, making each sensitive to changes in the other. The advantage of keeping components decoupled and separate is that a programmer can confidently make modifications to one component without affecting other components. Testing can be limited to the changed component, rather than a combination of components, which quickly leads to an exponentiation in the number of tests required.

In Eiffel, overloading is recognised as being problematic, so even this form is disallowed: routine arguments and local variables cannot overload names of class features.

C++ has another analogous form of hiding: a non-virtual function in a derived class hides a function with the same signature in an ancestor class. This hiding is explained in section 13.1 of the C++ ARM. This is confusing and error prone. Learning all these ins and outs of the language is extremely burdensome to the programmer, often being learnt only after falling into a trap. Java does not have this problem as everything is virtual, so a function with the same signature will override rather than hide the ancestor function.

In order to overcome the effects of hiding, you can use the scope resolution operator '::'. The scope resolution operator of C++ provides an interesting twist to the above argument. Consider the following example from p16 of the ARM:

```
int g = 99;

int f(int g)  // hide the outer g.
{
    return g ? g : :: g;
        // return argument if it
        // is nonzero otherwise
        // return global g
}
```

This would be simpler if the compiler reported an error on the redefinition of `g` in the parameter list: the programmer would simply change the name of one of the entities with no need for the scope resolution operator:

```
int g = 99;

int f(int h)
{
    return h ? h : g;
}
```

With the introduction of namespaces in 1993, the '`::`' operator now resolves names in namespaces. For example `A::x`, means the entity x in namespace A. Above `::g` means the entity `g` in the global namespace. Since declarations in a namespace are really just members of a fixed structure, it would have been cleaner to just use the access operator '.', and avoid the ugly scope resolution operator.

Java does not provide a scope resolution operator. However, there are no globals, so the only case where the above is a problem is between class members, and method parameters or locals.

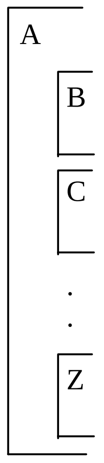
Java does have a similar problem though. The problem is with *shadowed variables*. With shadowed variables, a variable named *x* in a superclass can be hidden from the current class by another variable named *x*. You can still access both variables by the use of *this.x* and *super.x*, which are the equivalents of scope resolution. The ambiguity problem would have been better avoided altogether by reporting a duplicate identifier.

Eiffel also has no globals, so a construct such as namespaces is not needed. Eiffel does not allow name clashes: you must either change the name of one of the entities, or when combining classes with inheritance, use a **rename** clause. With this scheme there is no need for scope resolution or 'super' operators, making the imperative part of the language simpler, by using declarative techniques.

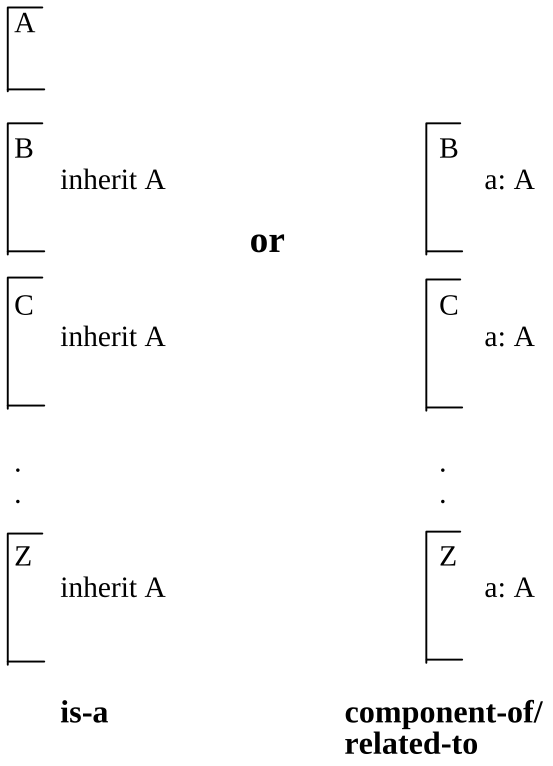
3.10 Nested Classes

Simula provided textually nested classes similar to nested procedures in ALGOL. Textual (syntactic) nesting should not be confused with semantic nesting, nor static modelling with dynamic run-time nesting. Modelling is done in the semantic domain, and should be divorced from syntax; you do not need textually nested classes to have nested objects. Nested classes are contrary to good object-oriented design, and the free spirit of object-oriented decomposition, where classes should be loosely coupled, to support software reusability.

Instead of tightly coupled environments:



You should decouple depending on the modelling requirements:



This is a more flexible arrangement, both in terms of modelling and program maintenance.

There are two problems with nested classes: firstly, the inner class is dependent on the outer class, and so is not reusable, contrary to good object-oriented design, where classes are independent; secondly, the inner class has access to the implementation of the outer class, so implementation hiding is violated. Where access to a class's implementation is needed, you should use inheritance, but note this models the is-a relationship, not the component-of relationship that nested classes do.

Semantic nesting is achieved independently of textual nesting. In object-oriented design all objects should interact only via well defined interfaces, but objects of a class that is textually nested in another class have access to the outer object without the benefit of a clean interface. C avoided the complexity of nested functions, but C++ has chosen to implement this complexity for classes, which is of less use than nested functions, and is contrary to good object-oriented design.

Pascal and ALGOL programmers sometimes use nested procedures in order to group things together, but nested procedures are not necessary, and if you want to use a nested procedure in another environment, you have to dig it out of where it is and make it global, which is a maintenance problem. If the procedure uses locals from the outer environment, you have more problems. You will have to change these to parameters, which is a cleaner approach anyway, and you will probably have to unindent all the text by one or more levels. Textually nested classes have worse problems.

Semantically, OOP achieves nesting in two ways: by inheritance and object-oriented composition. Modelling nesting is achieved without tight textual coupling. Consider a car. In the real world the engine is embedded in the car, but in object-oriented modelling embedding is modelled without textual nesting. Both car and engine are separate classes: the car contains a reference to an engine object. This allows the vehicle and engine hierarchy to be independently defined. Engine is derived independently into petrol, diesel, and electric engines. This is simpler, cleaner and more flexible than having to define a petrol engine car, a diesel engine car, etc., which you have to do if you textually nest the engine class in the car. In the real world you can change the cars engine, so it does not even make sense to tightly couple the car and the engine.

In C++, not only can classes be nested within other classes, but also within functions, thereby tightly coupling a class to a function. This confuses class definition with object declaration. The class is the fundamental structure in object-oriented programming and nothing has existence separate from a class (including globals).

Neither Java, nor Eiffel provide nested classes, and yet everything you can model in C++, you can also model in these languages, without the problems associated with textual nesting.

Chapter 18 of [Madsen 93] provides very good insights about modelling; classification and composition are the means to organise complexity in terms of hierarchies. [Madsen 93] enumerates four kinds of composition: whole-part composition, reference composition, localisation, and concept composition. They say that these are not altogether independent as one composition relationship could fall into two or more categories. Whole-part composition models the car example above, where the engine is part of the car. Reference composition is illustrated where a person makes a hotel reservation. The person is not a part of the reservation, but the reservation references the person. [Madsen 93] can be consulted for definitions of localisation and concept composition.

As examples can be given of composition that can be modelled in terms of more than one of the categories of composition, it is better not to provide direct modelling of this in the programming language; your opinion might later change. BETA does have mechanisms for modelling the whole-part composition as embedded objects, and reference as references. However, this is quite different to textual nesting. There is no real need to support these different categories in your programming language. It is more important for the analyst to be cogniscent of these different flavours so that he can recognise different kinds of composition in the problem domain.

3.11 Global Environments

There are two important properties of globals: firstly, a global is visible to the whole program, which is a compile-time view; and secondly, a global is active for the entire execution of a program, which is a run-time property. The first property is not desirable in the object-oriented paradigm, as will be explained below. The second property can easily be provided. The life of any entity is the life of the enclosing object, so to have entities that are active for the whole execution of the program, you create some objects when the program starts, which don’t get deallocated until the program completes.

The global environment provides a special case of nested classes. When classes are nested in a global environment, dependencies can arise that make the classes difficult to decouple from the original program, and therefore not reusable, by themselves. You might be forced to relocate a large amount of the global environment as well. There are also problems with the related mechanisms of header files and namespaces. Even if a class is not intended for use in another context, it will benefit from the discipline of object-oriented design. Each class is designed independently of the surrounding environment, and relationships and dependencies between classes are explicitly stated.

In C++ functions can change the global environment, beyond the object in which they are encapsulated. Such changes are side-effects that limit the opportunity to produce loosely-coupled objects, which is essential to enable reusable software. This is a drawback of both global and nested environments. A good OO language will only permit routines in an object to change its state.

Removing the global environment is trivial: simply encapsulate it in an object or set of objects. The previously global entities are then subject to the discipline of object-oriented design; globals circumvent OOD. Objects can also provide a clean interface to the external environment, or operating system, without loss of generality, for a negligible performance penalty. Classes are independent of the surrounding environment, and the project for which they were first developed, and are more easily adaptable to new environments and projects.

Java has removed globals from the language altogether. Eiffel is another example of a language where there are no globals. Both these languages show that globals are not needed for, and even detrimental to the development of large computer systems.

In concurrent and distributed environments you are better off without globals. In a distributed environment, the global state of the system may be impossible to determine. In order to develop distributed systems, you cannot have globals. Similarly with concurrent environments, problems arise when two or more process threads access shared resources at the same time. Shared resources should only be accessed via an object which manages the resource, and prevents contention for the shared resource. Such a resource should not be a global.

3.12 Polymorphism and Inheritance

Inheritance provides a textually decoupled form of subblock. The scope of a name is the class in which it occurs. If a name occurs twice in a class, it is a syntax error. Inheritance introduces some questions over and above this simple consideration of scope. Should a name declared in a base class be in scope in a derived class? There are three choices:

1) Names are in scope only in the immediate class but not in subclasses. Subclasses can freely reuse names because there is no potential for a clash. This precludes software reusability. Since subclasses will not inherit definitions of implementation, case 1 is not worth considering.

2) The name is in scope in a subclass, but the name can be overloaded without restriction. This is closest to the overloading of names in nested blocks. This is C++'s approach. Two problems arise: firstly, the name can be reused so the inherited entity is unintentionally hidden; secondly, because the new entity is not assumed to have any relationship to the original, its signature cannot be type checked with the original entity. Since consistency checks between the superclass and subclass are not possible, the tight relationship that inheritance implies, which is fundamental to object-oriented design, is not enforced. This can lead to inconsistencies between the abstract definition of a base class, and the implementation of a derived class. If the derived class does not conform to the base class in this way, it should be questioned why the derived class is inheriting from the base class in the first place. (See the nature of inheritance.)

3) The name is in scope in the subclass, but can only be overridden in a disciplined way to provide a specialisation of the original. Other uses of the name are reported as duplicate name errors. This form of overriding in a subclass ensures the entity referred to in the subclass is closely related to the entity in the ancestor class. This helps ensure design consistency. The relationship of name scope is not symmetric. Names in a subclass are not in scope in a superclass (although this is not the case in dynamically typed languages such as Smalltalk). In order to provide the consistent customisation of reusable software components, the same name should only be used when explicitly redefining the original entity. The programmer of the descendant class should indicate that this is not a syntax error due to a duplicate name, but that redefinition is intended, (the suggested keyword `override` has already been covered in the virtual section.) This choice ensures that the resultant class is logically constructed. This might seem restrictive, but is analogous to strong typing, and makes inheritance a much more powerful concept.

3.13 Type Casts

"Syntactically and semantically, casts are one of the ugliest features of C and C++." not my words or any other detractor of C++, but from [Stroustrup 94].

Mathematical functions map values from one type to values of another type. For example arithmetic multiplication maps the type 'pair of integers' to an integer:

```
Mult:INTEGER x INTEGER -> INTEGER
```

A language type system enables a programmer to specify which mappings make sense. Like functions, type casts map values of one type onto values of another type, but this *forces* one type to another, against the defined mappings, undermining the value of the type system. A strongly typed language with a well defined type system does not need casts: all type to type mapping is achieved with functions that are defined within the type system; no casts outside the type system are needed.

Type casts have been useful in computer systems. Sometimes it is required to map one type onto another, where the bit representation of the value remains the same. Type casts are a trick to optimise certain operations, but provide no useful concept that general functions don't provide. In many languages, the type system is not consistently defined, so programmers feel that type casts are necessary, or the language would be restrictive.

An example often used in programming is to cast between characters and integers. Type casts between integers and characters are easily expressed as functions using abstract data types (ADTs).

TYPE
    CHARACTER

FUNCTIONS
    ord: *CHARACTER* -> *INTEGER*
    // convert input character to integer
    char: *INTEGER* /-> *CHARACTER*
    // convert input integer to character

PRECONDITION
    // check *i* is in range
    pre char (*i*: *INTEGER*) =
        0 <= *i* **and** *i* <= **ord** (*last character*)

The notation '->' means every character will map to an integer. The partial function notation '/->' means that not every integer will map to a character, and a precondition, given in the **pre char** statement, specifies the subset of integers that maps to characters. Object-oriented syntax provides this consistently with member functions on a class:

i: *INTEGER*
ch: *CHARACTER*

```
i := ch.ord
// i becomes the integer value of the character.
ch := i.char
// ch becomes the character corresponding to i.
```

but a routine char would probably not be defined on the integer type so this would more likely be:

```
ch.char (i)
// set ch to the character corresponding to i.
```

The hardware of many machines cater for such basic data types as character and integer, and it is probable that a compiler will generate code that is optimal for any target hardware architecture. Thus many languages have characters and integers as built in types. An object-oriented language can treat such basic data types consistently and elegantly, by the implicit definition of their own classes.

Another example of type conversion is from real to integer; but there are several options. Do you truncate or round?

```
TYPE
    REAL

FUNCTIONS
    truncate: REAL -> INTEGER
    round: REAL -> INTEGER

    r: REAL
    i: INTEGER

    i := r.truncate
    // i becomes the closest integer
    // <= r
    i := r.round
    // i becomes the closest integer to r
```

Again many hardware platforms provide specific instructions to achieve this, and an efficient object-oriented language compiler will generate code best optimised for the target machine. Such inbuilt class definitions might be a part of the standard language definition.

3.14 RTTI and Type casts

Since the second edition of this critique in 1992, C++ added Run-Time Type Information (RTTI) in March 1993. This is a good and necessary feature, and a discussion of it helps clarify the notion of casts.

[P&S 94] makes a case against rejecting all programs that are not statically type correct. If a program is shown to be statically type correct, its type correctness is *guaranteed*, but static type checks can reject a class of programs that are otherwise type valid.

List classes are an example of where static type checking can reject a valid program. A list class can contain objects of many different types. Genericity and templates allow constructions such as *list of objects*, *list of animals*, etc. These are types built from the generic *list* class.

In the list of animals, you might know that squirrels occur in even numbered slots in the list. You could then assign an even numbered list element to a variable of type squirrel. Dynamically, this is correct, but statically the compiler must reject it as it does not know that only squirrels occur in even locations in the list.

Things aren't always this simple. The programmer probably won't know the pattern of how particular animals are stored in the list. Consider a vet's waiting room. The vet might view his waiting room as being the type: *list of animals*. Calling in the first animal from the waiting room, it is important to know whether the animal is a cat or a hamster if the vet is to perform an operation on the animal. For many such cases object-oriented dynamic binding and polymorphism will suffice, so that the programmer does not have to know the exact type of the object, as long as the objects are sufficiently the same that the same operations can be applied, even though the implementations might be different.

However, this is not always sufficient, and sometimes it is important to know that you have retrieved a hamster from a list of animals.

For example, once our vet has performed the operation on the hamster or cat, he must know enough about their type to decide whether to now put the animal in the hamster cage, or the cat basket.

Casting can solve this problem, but it is a sledgehammer approach where much more elegant and precise solutions exist. [Stroustrup 94] notes: "The C and C++ cast is a sledgehammer."

Eiffel has such an elegant and precise solution called the *assignment attempt*, notated as '?=' instead of ':='. A simple example is:

```
waiting_room: LIST [ANIMAL]
fluffy: HAMSTER
h_cage: HAMSTER_CAGE

fluffy := waiting_room.first   -- error.

-- The above assignment will be rejected by the
-- compiler as type (fluffy) = HAMSTER and
-- ANIMAL is not a subtype of HAMSTER. Even
-- though we know that the animal will be a
-- HAMSTER, and the program is valid, static
-- type checking considers it invalid.

fluffy ?= waiting_room.first

-- If the first animal in the waiting room is
-- indeed a HAMSTER, then fluffy will refer
-- to that animal, else fluffy will be Void.

if fluffy /= Void then
    h_cage.put (fluffy)
end
```

The Eiffel *assignment attempt* provides a precise and elegant solution to the dynamic type problem. Since the assignment attempt has the desired effect of by-passing static type checking and leaving it to run time, type casting is not needed.

If you want to be as flexible as Smalltalk, you could use assignment attempt instead of straight assignment everywhere, but as this invokes run time type checks, and you must check for *Void* references, there is a large overhead to assignment attempt over straight assignment. This shows that not only is static typing important for proving compile-time correctness, but also for run-time efficiency. The only real effect of ?= as far as the programmer is concerned is that it suppresses the compiler's static type checking and puts in a run-time check.

As I said, C++ introduced Run-Time Type Information (RTTI) in March 1993. RTTI has the operator `dynamic_cast`, which achieves the same effect as the Eiffel assignment attempt. `dynamic_cast` returns a pointer to a derived class from a pointer to a base class if the object is an object of the derived class; otherwise it returns `0` (or should that be null? But `0` isn't really zero, but any bit pattern representing null).

In C++, the above assignment attempt would be coded:

```
fluffy =
    dynamic_cast<hamster*>
            (waiting_room.first());
```

A few observations. Wow! Eiffel uses an operator, and C++ uses a keyword. It should be noted though that in correctly designed programs, neither assignment attempt, nor `dynamic_cast` will be used very often. So this is a small point.

The second observation is that in C++ you must specify the type. In this example it is superfluous as the compiler can determine **type** (*fluffy*) = *HAMSTER*, as it does in Eiffel.

In C++ you can dynamically cast to any derived class from `hamster*` but that does not seem to gain anything. A second point is that you don't need to use `dynamic_cast` directly in an assignment, but can use it in a general expression. However, again it is stressed that run time casting should be so little used that this is of little advantage. Perhaps the only small advantage is the ability to be able to pass a dynamically cast pointer:

```
h_cage.put
    (dynamic_cast<hamster*>
            (waiting_room.first());
```

Looks good right? But remember, if the first animal out of the waiting room is not a hamster, but a rat, you get `0` (well null...etc) returned which will cause `h_cage.put()` to fail.

This shows that the use of `dynamic_cast` in an expression is not such a good idea, as it might cause the whole expression to fail.

Thus Eiffel's assignment attempt is safer and syntactically cleaner. And there is another reason for this remark: if you don't put the **if** *fluffy* /= *Void* **then** test in, either deliberately or because you forgot, then the precondition that is most likely in the Eiffel version of *h_cage.put* tests that the argument is not Void. If you deliberately left out the *Void* test, you will have included a **rescue** clause to handle this exception.

Although the Eiffel syntax '?=' for assignment attempt is cleaner, [Stroustrup 94] points out that such clean syntax would be inappropriate for C++. This is because the '?=' would be "difficult to spot" in C++'s otherwise clumsy syntax. This is why it is possible to use this neat notation in Eiffel, as Eiffel's syntax is much clearer, and since programmers will code small routines, the '?=' is not difficult to spot in an Eiffel program. The reasoning against '?=' in C++ is strange, since C already provides assignment operators like '`+=`' and '`-=`', which are just a small syntactic convenience.

Another RTTI feature is the `typeid` operator. [Stroustrup 94] warns against using this to determine program flow control based on type information. You should not use switch statements, but use dynamic binding on polymorphic (virtual) functions. This will need to be built into your style rules that programmers will hate, or you will end up having to fix the dirty deed after the fact, which adds to the expense of your software developments.

Eiffel has no built in operator to achieve this, so the object-oriented principle of using dynamic binding instead of switch statements is better enforced. Eiffel removes type identification from the language, but places it in the libraries in some routines built into the *GENERAL* class. So in Eiffel, it is harder to commit the bad programming practices that [Stroustrup 94] warns about.

3.15 New Type Casts

Not only did C++ introduce RTTI and `dynamic_cast` in March 1993, but also three more cast operators in November 1993. These operators are:

`static_cast<T>(e)`,
`reinterpret_cast<T>(e)`, and
`const_cast<T>(e)`.

Again for all these the specification of the <type> seems superfluous, as the compiler can derive that from the context. These casts just about cover all the cases where you would need to use C style casts.

[Stroustrup 94] indicates a desire to discard the C casts: "I intended the new-style casts as a complete replacement for the `(T)e` notation. I proposed to deprecate `(T)e`; that is, for the committee to give users warning that the `(T)e` notation would most likely not be part of a future revision of the C++ standard. ... However, that idea didn't gain a majority, so that cleanup of C++ will probably never happen."

The bottom line to these sections on type casts comes again from [Stroustrup 94]: “In all cases, it would be better if the cast - new or old - could be eliminated.” It can! Use Eiffel or another one of the languages in which the type system is more cleanly defined.

3.16 Java and Casts

Unfortunately, Java needs casts in the above examples, but has improved the situation: “Not all casts are permitted by the Java language. Some casts result in an error at compile time. For example, a primitive value may not be cast to a reference type. Some casts can be proven, at compile time, always to be correct at run time. For example, it is always correct to convert a value of a class type to the type of its superclass; such a cast should require no special action at run time. Finally, some casts cannot be proven to be either always correct or always incorrect at compile time. Such casts require a test at run time. A ClassCastException is thrown if a cast is found at run time to be impermissible.” - from the Java Language Specification.

3.17 ‘.’ and ‘->’

The ‘.’ and ‘->’ member access syntax came from C structures, and illustrates where the C base adversely affects flexibility. Semantically both access a member of an object. They are, however, operationally defined in terms of how they work. The dot (‘.’) syntax accesses a member in an object directly: ‘x.y’ means access the member y in the object x.

```
OBJ x; // declare object x of
       // class obj
       // with a member y.
x.y;   // access y in object x
       // directly
x->y;  // syntax error “. expected”
```

The specific error is:

```
error: type 'OBJ' does not have an
   overloaded member 'operator ->'
error: left of '->y' must point
   to class/struct/union
```

The ‘->’ syntax means access a member in an object referenced by a pointer: ‘`x->y`’ (or the equivalent `*(x).y`) means access the member `y` in the object pointed to by `x`.

```
OBJ *x; // declare a pointer x to an
        // object of class obj.
x->y;   // access y via pointer x
x.y;    // syntax error “-> expected”
```

The specific error is:

```
error:'.OBJ::y' : left operand points
   to 'class', use '->'
```

In these examples, ‘what’ is to be computed is “access the element y of object x.” In C++, however, the programmer must specify for every access the detail of ‘how’ this is done. That is the access *mechanism* to the member is made visible to the programmer, which is an implementation detail. Thus the distinction between ‘.’ and ‘`->`‘ compromises implementation hiding, and very seriously the benefit of encapsulation. We will see in the section on inlines how the visible difference of access mechanisms between constants, variables and functions also breaks the implementation hiding principle, and how the burden is on the programmer to restore hiding, rather than fix the language.

The compiler could easily restore implementation hiding by providing uniform access and remove this burden from the programmer, as in fact most languages do. The major benefit of implementation hiding is that if the implementation changes, the effect is contained within the class itself; not manifest beyond the interface. Where implementation hiding is broken, the effects of implementation change become visible, and this reduces flexibility.

For example, if the ‘`OBJ   x`’ declaration is changed to ‘`OBJ  *x`’, the effect is widespread as all occurrences of ‘`x.y`’ must be changed to ‘`x->y`’. Since the compiler gives a syntax error if the wrong access mechanism is used, this shows that the compiler already knows what access code is required and can generate it automatically. Good programming centralises decisions: the decision to access the object directly or via a pointer should be centralised in the declaration. So again, C++ uses low level operators, rather than the high level declarative approach of letting the compiler hide the implementation and take care of the detail for us.

Java only supports the dot form of access. The ‘`->`‘ form is superfluous. Java objects are only accessed by reference; there are no embedded objects.

Eiffel provides a more interesting case. In Eiffel an optimisation is provided as an object can be expanded in line in another object, in order to save a reference. Eiffel calls such objects **expanded** objects. There is still no need for explicit dereferencing. The compiler knows exactly whether the object is expanded or referenced, and thus the dot accessor is used for both, so uniform access is provided, and the access mechanism is hidden. This makes the program more malleable, as the programmer can later change an object to expanded, and not have to worry about changing every ‘`->`‘ to a dot. Conversely, if expansion turns out to be inappropriate, as in the case of a circular reference, then the expanded status of the object can be removed from the declaration, without having to change another single line of code. Thus Eiffel preserves the implementation hiding principle, which results in convenience for the programmer.

There is even more to Eiffel's scheme, which is particularly relevant to concurrent and distributed processing. Meyer points out in [Meyer 96c] that the form *x.f* means passing the message *f* to the object *x*. *x* may be anywhere on the network. In other words, *x* might not be a reference that is implemented by an underlying C pointer, but it may be a network address, for example a URL.

3.18 Anonymous parameters in Class Definitions

C++ does not require parameters in function declarations to be named. The type alone can be specified. For example a function `f` in a class header can be declared as `f (int, int, char)`. This gives the client no clue to the purpose of the parameters, without referring to the implementation of the function. Meaningful identifiers are essential in this situation, because this is the abstract definition of a routine; a client of the class and routine must know that the first `int` represents a 'count of apples', etc. It is true that well known routines might not require a name, for example `sqrt (int)`. But this is not appropriate for large scale software development.

The use of anonymous parameters handicaps the purpose of abstract descriptions of classes and members: to facilitate the reusability of software. This is covered in more detail in the section on 'Reusability and Communication'. Program text captures the meaning of the system for some future activity, such as extension or maintenance. To achieve reusability, communication of intent of a software element is essential.

Names are not strictly necessary in programming. Naming exists to help the human reader identify different entities within the program, and to reason about their function. For this reason naming is essential; without it, development of sophisticated systems would be nearly impossible. Some languages access parameters by their address (position) in the parameter list ($1, $2, etc). This is unsatisfactory, even for shell scripts. Anonymous parameters can save typing in a function template, but then programming is not a matter of convenience as it is inconvenient for later readers. The redundancy is beneficial and saves later programmers having to look up the information in another place. A real convenience in function templates would be that abstract function templates be automatically generated from the implementation text (see header files for more details).

Anonymous parameters illustrate the link between courtesy and safety issues in programming. Due to pressure of work, a client programmer might wrongly guess the purpose of a parameter from the type. The failure of the original programmer to provide a courtesy has caused a client programmer to breach safety. However, the client programmer will probably be blamed for not taking due care. An interface client must know the intention of the interface for it to be used effectively.

Both Java and Eiffel do away with the distinction between a function definition and declaration. The first reason for this is that you don't need forward declarations, as entities can be referenced before they are declared. The second reason is that in Eiffel, there are tools to automatically extract abstract interface definitions from the main code.

3.19 Nameless Constructors

Multiple constructors must have different signatures, similar to overloaded functions. This precludes two or more constructors having the same signature. Constructors are also not named (apart from the same name as the class), which makes it difficult to tell from the class header the purpose of the different constructors. Constructors suffer from all of the problems described with regards to overloaded functions. Firstly, it would be easy to mark routines as constructors, for example:

```
constructor make (...)...
constructor clone (...)...
constructor initialise (...)...
```

where each constructor leaves the object in valid, but potentially different states. Named constructors would aid comprehension as to what the constructor is used for in the same way as function names document the purpose of a function. Secondly, named constructors would allow multiple constructors with the same signature. Thirdly, it is easier to match up an object creation with the constructor actually called. Fourthly, the compiler could check the arguments given in the invocation to the constructor signature.

Java's constructor scheme is the same as C++. Eiffel allows a series of *creation* routines. These are indeed independently named as suggested above.

Eiffel has another advantage in that creation routines can also be exported as normal routines which can be called to reinitialize an object. In C++ you cannot call a constructor, after the object is created.

3.20 Constructors and Temporaries

A 'return <expression>' can result in a different value than the result of <expression>. In section 6.6.3, the C++ ARM says: "If required the expression is converted, as in an initialisation, to the return type of the function in which it appears. This may involve the construction and copy of a temporary object (S12.2)."

Section 12.2 explains: "In some circumstances it may be necessary or convenient for the compiler to generate a temporary object. Such introduction of temporaries is implementation dependent. When a compiler introduces a temporary object of a class that has a constructor it must ensure that a constructor is called for the temporary object."