C++??

A Critique of C++

2nd Edition

Ian Joyner

c/- Unisys - ACUS
115 Wicks Rd, North Ryde
Australia 2113
Tel: +61-2-390 1328 Fax +61-2-390-1391

ian@syacus.acus.oz.au

1. Introduction

The C++ programming language is becoming widely used. So it is important and timely to question its success. Two books are already published on the subject [Sakkinen 92] and [Yoshida 92]. This critique addresses the following questions. How well does C++ implement object-oriented concepts? Can it easily implement small, quick projects? Does it scale up well for large projects? Does it support or hinder good programming practices? As a result, does it ease the production of quality software? What is the relationship between a language, compiler and software developers; and between the language, compiler and the target system? This last question addresses issues of correctness, compatibility, portability, and efficiency.

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 costly burden? The 'fine line' is a result of poor language definition. The C++ standardisation committee warns "C++ is already too large and complicated for our taste" [X3J16 92].

While it is true that C++ is immediately usable by many C programmers, and many see this as a strength, the C base is C++'s greatest weakness. This is the engineering compromise that C++ devotees talk about. 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 critique criticises C++ in its own right, without comparison to other languages. Section 2 considers the role of a programming language. Section 3 examines some specific aspects of C++. Section 4 examines the general role of language. Section 5 is a short comment on writing. Section 6 looks specifically at C. The conclusion examines where C++ has left us, and considers the future. The approach taken is to criticise specific aspects of C++ and C. Each section tries to be self contained. It is expected that not everyone will agree with all of the sections. It is probably best to approach the paper, not by reading it entirely, but to read those sections that interest you. One section, however, is fundamental to the criticism of C++, that on virtual functions. This is also the most difficult to understand and technical section of the paper, but it is fundamental to the understanding of the weaknesses of C++.

Having said that, I hope that you find this critique useful, and enjoyable. If by any chance you do, 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. It should be critiqued with respect to those levels and roles. Historically, programming languages had a very 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. The organisation of projects also required tools external to the language and compiler, like 'make.' Object-oriented techniques have arisen to help in the analysis and design phases, and object-oriented languages to support the implementation phase of OO. Traditional, tried and tested but failed software practices are infiltrating the object-oriented world. 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.

C++ is an interesting experiment in adapting the advantages of object-orientation to a traditional programming language. Bjarne Stroustrup is to be applauded for having the insight to put the two technologies together. 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 limited analysis and design. The processes of analysis, design, and organisation, however, are still largely external to C++. Thus C++ has not realised the important advantages of object-orientation that will indeed lead to the economic production of software.

A language should not only be critiqued from a technical point of view, considering its syntactic and semantic features. It should also be critiqued from the viewpoint of its contribution to the entire software development process. It should enable communication between project members acting at different levels, from management, who have a requirement for the product, to testers, who must test the result. It should also enable communication between project members separated in space and time. Often one programmer is not responsible for a task over its entire lifetime.

The primary purpose of any language is communication. A programming language should support the exchange of ideas, intentions, and decisions between project members. A programming language 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.

A language definition should enable the development of integrated automated tools to support software development. For example, browsers, editors and debuggers. The compiler is another such tool. The role of a compiler is twofold. Firstly, to generate code 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 complete, valid and consistent. A compiler should perform semantics checking. This is checking that a program is internally consistent. Generating a system that has detectable inconsistencies is pointless.

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 the sentence is 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 language definition provides the framework that makes this role of the compiler possible.

Checking is often enabled by the specification of redundant information. Declarations are an example of redundancy that help check for misspellings. Declarations define the vocabulary of a program, ie the elements in its universe. The compiler uses redundant information for consistency checking, and strips it away to produce efficient executable systems. Type safety is another technique. Declarations also associate an entity with a type, to define the entities role. Typing ensures that you can't drink computers or switch on glasses of water. C++ is an improvement over C in type safety.

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 of 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.

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. A program description is constantly modified during development. Modifications often lead to inconsistencies and error. Languages and compilers that provide 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. Alerting programmers of potential errors earlier and interactively will shorten development times. Future languages should be defined very cleanly in order to enable such editor technology.

A programming language should provide a formal notation. During requirements analysis and design phases, formal and semi-formal notations are required. 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 programming, while programming languages are in general too low level. Both designers and programmers must compromise to fill the gap. Current notations provide difficult transition paths between stages. This 'semantic gap' contributes to errors and omissions between the requirements, design and implementation phases. Future programming languages will be an implementation extension of the high level notations used for requirements analysis and design. This 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 by the same syntax.

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. 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. They require that the syntax of a language 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.

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. 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. This is the problem domain. 'How' a computation is achieved is in the low-level machine-oriented domain. The conflict between these aspects recurs frequently throughout this critique. 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 some environments hinders portability to other platforms.

The industry should be moving towards these ideals. They 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. Its effectiveness, however, depends on the quality of its implementation.

It is relevant to ask if grafting OO concepts onto a conventional language realises the full benefits of OO? Perhaps a biblical quote can be considered: "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? 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.

2.1. 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 compromise them frequently. Safety concerns affect the *external* perception of the quality of the program. Failure to meet safety concerns results in unfulfilled requirements and program crashes.

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 courtesy for an airline to keep its fleet well maintained. This courtesy concern 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.

3. C++ Specific Criticisms

3.1. Virtual Functions

Polymorphism is a key concept of OOP. Virtual functions are one way to implement polymorphism. A language designer's choice is whether this should be specified in the parent or the inheriting class. Is it the decision of the designer of the parent or descendant class? Cases can be made for both. They are not mutually exclusive and can be catered for quite easily in an object-oriented language.

There are three options, corresponding to 'must not', 'can', and 'must' be redefined:

1) The redefinition of a routine is prohibited; descendant classes must use the routine as is.

2) A routine could be redefined. 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. This is polymorphism.

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 first option. 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 compiler should report a syntax error due to 'duplicate declaration'. This is logical as descendant classes are part of the same name space as classes they inherit from. The redeclaration of a name within the same scope should cause a name clash. 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, 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. B's designer has explicitly provided for this. 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. This is opposite to good modular design. This shows the unsafeness C++'s virtual mechanism. 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 routine forces a change in the base class. This has an effect on all other descendants of the base class, instead of the specific new requirement being localised to the new class. This is opposite 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.

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 specialized 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 subclasses, 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]

A further argument is that any statement should consistently have the same semantics. The object-oriented 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. A change in the declaration of f () will change 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 the program text is subject to perpetual change.

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. [Sakkinen 92] points out that a descendant class can redefine a private virtual function even though it cannot access that function in other ways. When the ancestor class calls the function it instead invokes the function in the descendant class.

Option 2

The second option should be left open for the programmers of descendant classes. In C++, however, the decision must be made in the base class. 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. 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.

Virtual, however, is the wrong mechanism for the programmer to deal with. A compilation system 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. The compiler should take care of such detail and so relieve the programmer.

Another problem in C++ is mistaken redefinition. The base class routine can be

redefined 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.

Eiffel and Object Pascal cater for this situation as the descendant class programmer is required to 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.

Option 3

The pure virtual function caters for the third option. The routine is undefined, the class is abstract and cannot be directly instantiated. A descendant class must define the routine 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 a criticism of the syntax.

Virtual is a difficult notion to grasp. The related concepts of polymorphism and dynamic binding, redefinition, and overloading are easier to grasp, being oriented towards the problem domain. Virtual routines are an implementation mechanism for polymorphism. Polymorphism is the ‘what’, and virtual is the ‘how’. Smalltalk and Objective-C 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. Interesting as underlying mechanisms might be for the theoretician or compiler implementer, 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 concurrency).

3.2. Pure Virtual Functions

As mentioned above, pure virtual functions provide a means of leaving a function undefined and abstract. A class that has such an abstract function cannot be directly instantiated. A non-abstract descendant class must define the function. The C++ pure virtual syntax is:

```
virtual void fn () = 0;
```

This leaves the reader to guess its meaning, even those well versed in object-oriented concepts. A better choice would have been a keyword such as ‘abstract’. Direct expression of concepts enhances communication, and the ease with which a language can be learnt. When learning a language it is often important to use the index of a text book. A keyword like ‘abstract’ would be easily found in an index. But what do you look for in the case of ‘= 0’? You might not even realise it is significant. It should have syntactic significance as abstract functions are a very important concept in object-oriented design. The C++ decision is in keeping with the C philosophy of avoiding keywords. This is often at the expense of clarity. A keyword would implement this concept more clearly. For example:

```
pure virtual void fn ();
```

or

```
abstract void fn ();
```

The mathematical notation used in C++ suggests that values other than zero could be used. What if the function is equated to 13? -

```
virtual void fn () = 13;
```

A function is either pure, or it is not. This to any analyst suggests a boolean state, which a single keyword conveys. A simple suggestion to fix this is to define ‘= 0’ as abstract:

```
#define abstract = 0
```

then

```
virtual void fn () abstract;
```

‘Pure virtual’ is also an abuse of natural language. It is a combination of words that are somewhat opposite in meaning. Pure means something that really is what it appears to be. For example pure gold. Virtual means something that appears to be what it actually is not. For example virtual memory. Perhaps virtual gold could be fools gold. As has been said before, virtual is a difficult concept to grasp. When it is combined with a word such as ‘pure’, the meaning becomes even more obscure. Modern language designers should be very careful in the vocabulary they choose.

3.3. The Nature of Inheritance

Inheritance is a close relationship. It provides a fundamental way to assemble software components. 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, and overloading by signature (see below) means that the compiler cannot check for this consistency. C++ does not realise this aspect of object-oriented design. This contributes to a wide and costly gap between analysis and design, and implementation.

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." [SJE91]. 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 [Weg90] (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 not break the assumptions of an ancestor class.

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 system that has never existed before.

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. Certainly, not very many reusable C++ libraries are available, which suggests that C++ might not support reusability as well as possible. C++ fails to provide this fundamental goal of object-oriented design and programming.

3.4. Function Overloading

C++ allows functions to be overloaded if the arguments in the signature are of different types. Such 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 might want to accept two or more parameters of the same type.)

The above shows that in most cases, the object-oriented paradigm can consistently express 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.

The disadvantage is that C++ introduces some inconsistencies that the compiler cannot detect. If the programmer intends to redefine a virtual routine, but makes a mistake in the declaration of the function signature, the compiler will erroneously assume an overloaded function. Any calls to the function using one or other of the signatures will also fail to detect the inconsistency.

When calling the routine, if the programmer makes a mistake in supplying the actual

parameters, a C++ compiler cannot be specific about the error. It can only report that no function with a matching signature could be found. Programmers make this sort of mistake for subtle reasons, and it can be time consuming to pinpoint the parameter at fault. Secondly, the incorrect parameter might accidentally match, one of the other routines. In that case this error will be propagated into the production code, and could remain undetected a long time.

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. This is known to work and solves the above problems. 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.

3.5. Virtual Classes

If class D multiply inherits class A via classes B and C, then if D wants to inherit only a single copy of A, the inheritance of A must be specified as virtual in both B and C. This raises 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 implementer 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? Such 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.

3.6. Name overloading

Naming is fundamentally important in producing self-documenting software. Naming helps realise maintainable 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. Name overloading allows the same name to refer to two or more different entities. The problem is whether the resultant ambiguity is useful, and how to resolve it, as ambiguity weakens the power 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. 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 to be computed. Different classes might require different specifications 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. But use of the same name in the same context must be resolved. Errors can result from ambiguity. In this case the programmer needs to differentiate between entities in ways other than name alone. 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. For example a.mem and b.mem.

[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.

Block structured languages provide overloading by scoping. Scoping allows the same

name to be used in different contexts without clash or confusion. Nested blocks provide a subtle problem. Names in an outer block are in scope in inner blocks. Many languages, however, allow a name to be overloaded in an inner block. This does more than overload the name, it hides it. The use of a name in the inner block does not indicate any relationship with the same name in the outer block. Textually nested blocks 'inherit' named entities from outer blocks. Inheritance accomplishes this in object-oriented languages. Inheritance eliminates the need to textually nest entities, and also accomplishes loose coupling. Nesting makes entities tightly coupled.

Contrary to most high level languages, a name should not be overloaded while it is in scope. This inconveniently hides the outer declaration, and the programmer cannot access the outer entity. It is also error prone. The following example illustrates this:

```
{
  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 unmaintainable programs. This is because the inner block is tightly coupled to the outer block, and each is 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.

C++ has an analogous form of hiding. A non-virtual function in a derived class hides a function in an ancestor class. This hiding is explained in section 13.1 of the C++ ARM. This is a discrepancy with declaring multiple functions with the same name in the same class with different signatures. A function in the derived class will hide the functions of the ancestor class, rather than add its signature to the list of possible functions which can be called. This is confusing and error prone. Learning all these ins and outs of the language is extremely burdensome to the programmer. Often they will only be learnt after falling into a trap.

3.7. Polymorphism and Inheritance

Inheritance provides a form of name overloading similar to overloading in subblocks. 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. Subclasses will not inherit definitions of implementation. Therefore 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 unintentionally reused. 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 guaranteed. 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 overloaded 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 overloading 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 typeless languages such as Smalltalk). In order to provide the consistent customisation of reusable software components, the same name should only be used by 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. (This 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.

3.8. '.' and '->'

The '.' and '->' member access syntax came from C structures. It 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. For example '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 '->' syntax means access a member in an object referenced by a pointer. For example 'x->y' (or the equivalent *(x).y) means access the member y in the object pointer x refers to .

```
obj *x; // declare a pointer x to an
        // object of class obj.

x->y;   // access y via pointer x
x.y;    // syntax error "-> expected"
```

In this example, 'what' is to be computed is "access the element y of object x." In C++, however, the programmer must specify for every access the trivial detail of 'how' this is done. The compiler can easily remove this burden from the programmer, as in fact most languages do. Furthermore, this reduces flexibility as if the 'obj x' declaration is changed to 'obj *x', the effect is widespread as all 'x.y' must be changed to 'x->y'. Since the compiler gives a syntax error if the wrong access is used, this shows it 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.

3.9. Anonymous parameters in Class Definitions

C++ does not require parameters in function templates 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. 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. A compiler strips away this level of communication, producing a machine executable entity. Languages and compilers that perform less than optimal translations should not penalise careful production of semantic entities. But neither should a language definition allow less than optimal expression to the human reader. Languages do not have to be cryptic to achieve efficiency. In fact cryptic languages impair efficiency, as they make it harder for the programmer to develop efficient systems, and furthermore, they make it harder for automatic code optimisers.

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 naming, 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 quite unsatisfactory, even for shell scripts. Anonymous parameters can save typing in a function template, but then programming is not a matter of convenience. This 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. Thus the failure of the original programmer to provide a courtesy has

caused a later programmer to breach safety. An interface client must know the intention of the interface for it to be used effectively.

3.10. Nameless Constructors

Multiple constructors can 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). This makes it difficult to discern from the class header the purpose of the different constructors. It is difficult to match an object creation with the called constructor. Constructors suffer from all of the problems described with regards to functions with the same name but different signatures. 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.

3.11. 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."

A note says "The implementation's use of temporaries can be observed, therefore, through the side effects produced by constructors and destructors."

Putting this together, creation of a temporary is implementation dependent, so might or might not be done. If a temporary is created, a constructor is called as a side effect, which can change the state of the object. Different C++ implementations could therefore return different results for the same code.

3.12. Optional Parameters

Optional parameters that assume a default value according to the routines declaration are supposed to provide a shorthand notation. Shorthand notations are intended to speed up software development. Such shorthand notations can be convenient in shell scripts, and interactive systems. In large scale software production, however, precision is mandatory, and defaults can lead to ambiguities and mistakes. With optional parameters the programmer could assume the wrong default for a parameter. More importantly, optional parameters undermine type safety. The type of a function is defined by the composition of its input types, and its output type:

```
f: T1 x T2 x T3...  -> T4
```

The entire signature determines the type of the function, not just the return type. Optional parameters mean that C++ is not type safe, and that the compiler cannot check that the parameters in the call exactly match the function signature.

Furthermore, they do not provide a great deal of convenience. If a routine has five parameters, the last three of which are optional, and caller wants to assume the defaults for parameters 3 and 4, but must specify parameter 5, then all five parameters must be specified. A better scheme would be to have a 'default' keyword in function calls:

```
f (a, b, default, default, e);
```

Other means, already in the language, can easily provide this mechanism. For example, a call to another (possibly inline) function could provide the defaults for the optional parameters. This not only provides the convenience of optional parameters, but is more powerful. Any parameter or combination can be filled in with any combination of defaults, not just the last parameters. Multiple intermediate routines can provide multiple sets of defaults.

3.13. Bad deletions

The following example is given on p.63 in the C++ ARM as a warning about bad deletions that cannot be caught at compile-time, and probably not immediately at run-time:

```
p = new int[10];
p++;
delete p; // error
p = 0;
delete p; // ok
```

One of the restrictions of the design of C++ is that it must remain compatible with C. This results in examples like the above, that are ill-defined language constructs, that can only be covered by warnings of potential disaster. Removal of such language deficiencies would result in loss of compatibility with C. This might

be a good thing if problems such as the above disappear. But then the resultant language might be so far removed from C that C might be best abandoned altogether.

3.14. Local entity declarations

Declaring an entity close to where it is used, has both advantages and disadvantages. It is convenient, but can make a routine appear more complex and cluttered. A problem is that an identifier can be mistakenly overloaded within a nested block in a function, with the resultant problems covered in the sections on name overloading and nesting. C does not have nested routines or blocks so does not have this problem. ALGOL uses this simple form of name overloading. (A block in the ALGOL sense contains both declarations and instructions.)

The ARM explains problems of local declarations with branching, which shows the complications in intermingling declarations and instructions. Caveats cannot make up for or fix faulty language definition.

The C++ FAQ [Cline] (Q83) is unclear on this point (although it is mostly excellent), claiming that an object is created and initialised at the moment it is declared. This only applies to auto, in stack objects. Dynamic entities are not created and initialised until they are the subject of a 'new' instruction. In well written object-oriented software, routines will be small, typically performing one atomic action per routine.

Small routines that implement atomic operations are fundamental to loose coupling. For example, a base class that provides a single routine that logically performs operations A and B, is not useful to a subclass that needs to provide its own implementation of B, but does not want to change A. The descendant must reimplement the logic of both A and B, missing an opportunity to reuse the logic of A. Tight coupling reduces flexibility. Splitting A and B into different routines accomplishes loose coupling, and therefore flexibility. Efficiency is also attained without the mess of local entity declarations. Good design and clean modularisation achieve efficiency, as the entities which would be locals to a block in C++ are only created when the routine is entered.

3.15. Members

Care should be taken with the C++ use of the term member. In general use, an object is a member of a class. This corresponds to members in set theory. But in C++, the term member means a data item, or function of the class. This ambiguity could have easily been avoided.

3.16. Friends

Friends are a mechanism to override data hiding. Friends of a class have access to its private data. Friend is a 'limited export' mechanism. Friends have three problems:

1) They can change the internal state of objects from outside the definition of the class.
2) They introduce extra coupling between components, and therefore should be used sparingly.
3) They have access to everything, rather than being restricted to the members of interest to them.

Friends are useful, and a case can be made for shades of grey between public, protected and private members. Multiple interfaces to a class provide the functionality of friends and avoid the above problems. Each interface to the class can be exported to everything, or selected classes only. A selective export mechanism is more general than public, private, protected and friend, and explicitly documents the couplings between entities in the system. Selective export specifies not only that a member is exported but to which classes it is exported.

One reason given for friends, is that they allow more efficient access to data members than a function call. The way C++ is often used is that data members are not put in the public section, because this breaks the data hiding principle.

Data hiding is better described as 'implementation hiding'. Only a classes abstract functional interface should be visible to the outside world. That is data members can be exported, but are viewed externally as functional entities. This is because, when used in expressions, functions and variables have no semantic difference. They both return values of a given type. (See fn () for an explanation of why variables and functions are best regarded as similar entities.) (See also Marshall Cline's explanation of friends in the FAQ for further clarification of the friend concept.)

The Cambridge Encyclopedia of Language has an interesting point about public and private names. It says "Many primitive people do not like to hear their name used, especially in unfavourable circumstances, for they believe that the whole of their being resides in it, and they may thereby fall under the influence of others. The danger is even greater in tribes (in Australia and New Zealand, for example), where people are given two names - a 'public' name, for general use, and a 'secret' name, which is only known by God, or to the closest members of their group. To get to know a secret name is to have total power over its owner."

3.17. Static

The word 'static' is confusing in C++. Page 98 of the C++ Annotated Reference Manual (ARM) mentions this confusion and gives two meanings. Firstly, a class can have static

members, and a function can have static entities. The second meaning comes from C, where a static entity is local in scope to the current file. The choice of different keywords would easily solve this trivial problem. There is also a third more general meaning that objects are statically or automatically allocated and deallocated on the stack when a block is entered and exited, as opposed to dynamically allocated in free space.

Static class members are useful. Page 181 of the ARM states that statics reduce the need for global variables. It is good to reduce global variables, but the C syntax obscures the purpose.

Entities declared in functions can also be static. These are not needed in an object-oriented language. The reason and history is this. ALGOL has the notion of 'OWN' locals in blocks. The semantics of an OWN entity is that when a block is exited, the value of the OWN is preserved for the next entry to the block. I.e. the value is persistent. The implementation is that at compile time, the OWN entity is limited in scope to the block, but at run time, it is located in the global stack frame. The same instance of the variable is used in all invocations of the procedure, rather than each invocation using separate local storage on the stack. This causes complication in recursion.

Simula's designers generalised the ALGOL notion of block into class, and so object-orientation was born. Instead of discarding a class block on exit, it is made 'persistent'. Declarations within the class block are persistent, and therefore provide the functionality of static and OWN. Classes are more flexible than statics. Statics are persistent in the same way as globals, ie for the duration of the program. Class member lifetime is governed by the lifetime of the object. Object-oriented languages do not need OWNs or statics.

3.18. Union

Union is another construct that is superfluous in OOP. Similar constructs in other languages are recognised as problematic. For example, FORTRAN's equivalences, COBOLs REDEFINES, and Pascal's variant records. When used to overload memory space these force the programmer to think about memory allocation. Recursive languages use a stack mechanism that makes overloading memory space unnecessary, as it is allocated and deallocated automatically for locals when procedures are entered and exited. The compiler and run time system automatically allocate and deallocate storage as required, ensuring that two pieces of data never clash for the same memory space. This is essential so that the programmer can concentrate on the problem domain, rather than machine oriented details. When union is used similarly to FORTRAN's equivalences it is not needed.

Union is also not needed to provide the equivalent to COBOL REDEFINES or Pascal's variants. Inheritance and polymorphism provide this in OOP. A reference to a superclass can also be used to refer to any subclass, and thus provides the same semantics as union, only in a type safe manner, as the alternatives can never be confused. An object reference is implicitly a union of all subclasses.

3.19. 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. Semantic nesting is achieved independently of textual nesting. In object-oriented design all objects should interact only via well defined interfaces. 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.

OOP achieves nesting in two ways: by inheritance and object-oriented composition. Thus modelling nesting is achieved without tight textual coupling. For example, consider a car. We know in the real world that the engine is embedded within the car. In object-oriented modelling, however, this embedding is modelled without textual nesting. Both car and engine are separate classes. The car contains a reference to an engine object. This also allows the vehicle and engine hierarchy to be independently defined. Engine is derived independently into petrol, diesel, and electric engines. This is simpler 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. Other examples can also be structured without textual nesting, and no loss of generality.

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 class (including globals). C++ is confused as to whether it is procedure-oriented or object-oriented.

3.20. Global Environments

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 that environment, and therefore not reusable. 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. It is simply encapsulated in an object or set of objects of its own. Therefore global entities are subject to the discipline of object-oriented design. Having globals in a system circumvents OOD. Objects can also provide a clean interface to the external environment, or operating system, without loss of generality, for a negligible performance penalty. Thus classes are independent of a surrounding environment, and the project for which they were first developed, and are more easily adaptable to new environments and projects.

3.21. Header Files

In C++ a class interface must be maintained separately from its body. While an abstract interface should be distinct from a concrete implementation, the interface and implementation can both be derived from one source. In C++ though, programmers must maintain the two sets of information. Replicated information has well known drawbacks. In the event of change, both copies must be updated. This can lead to inconsistencies that must be detected and corrected. Tools can automatically extract abstract class descriptions from class implementations, and guarantee consistency.

The programmer must also use #includes to manually import class headers. #include is an old and unsophisticated mechanism to provide modularity. #include is a weak form of inheritance and import. C++ still uses this 30 year old technique for modularisation, while other languages have adopted more sophisticated approaches, for example, Pascal with Units, Modula with modules, Ada with packages. In Eiffel the unit of modularisation is the class itself, and includes are handled automatically. The OOP class is a more sophisticated way to modularise programs. Inheritance implements reusability and modularisation, so #include is superfluous.

Another problem is that if header A includes header B, and header B includes header A a circular dependency occurs. The same problem occurs if header A includes headers B and C, and header B also includes header C. A simple but messy fix in all headers solves this problem:

```
#ifndef thismod
#define thismod

... rest of header
#endif
```

Headers show how C++ addresses the problem of independent modules by a non-object-oriented approach that is sub-optimal; the programmer must supply this bookkeeping information manually. A class interface is equivalent to a module header. A module header contains data and routines exported to other modules. This is exactly the purpose of the class interface. A class definition contains all knowledge of component classes and their dependencies (inheritance and client) in the class text. Dependency analysis is derivable from the class text. Tools like ‘make’ can be integrated into the compiler itself, and the errors and tedium encountered in the use of ‘make’ are avoided. #includes relate to the organisation and administration of a project. Rational language design eliminates such bookkeeping mechanisms.

A traditional system is assembled by combining modules. An object-oriented system is assembled by combining classes. Modules are a primitive form of classes. Classes are more sophisticated. They express more precisely relationships with other classes. C++ #includes and modules have problems. This primitive method is not required in an object-oriented language.

3.22. Class Interfaces

Section 9.1c of the C++ ARM points out that C++ has no direct support for “interface definition” and “implementation module”. In a C++ class definition, all private and protected members must be included in the public text of the class. The ARM points out that whenever the private or protected parts are changed, the whole program must be recompiled. Further to what the ARM says, all modules that are dependent on the header file must be recompiled, even though the private and protected members do not affect other modules. Private members should not be in the abstract class interface, as this exposes implementation details to programmers of other modules.

3.23. Class header declarations

C’s syntax for function declarations is [<type>] <identifier> (<parameters>). For (a very simple) example:

```
class C
{
   a ();
   b ();
```

```
    int c ();
    d ();
    char e ();
    virtual void f ();
}
```

To find an identifier in this layout, the eye must trace a course around the type specifications. This is a tiring activity. The eye has a greater chance of missing the sought identifier, and the programmer must resort to using the search function of a text editor to help out.

Other languages place the entity names first. For example:

```
class C
{
    a ();
    b ();
    c () int;
    d ();
    e () char;
    f () virtual void;
}
```

To those used to the ALGOL and FORTRAN style of type first, this seems backwards. But name first is logical as a real world example illustrates. Imagine if a dictionary is published, and the keywords are not placed first, but rather the entry order is -

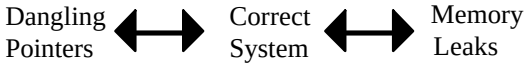
```
noun /obvrzen/ obversion, the act or
     result of obverting
```

Such a dictionary would not sell many copies, unless the marketers managed to fool many people that the explanation of the meaning was more correct because the order of layout was mysteriously magical. This example illustrates how important subtle syntax decisions are, and why PASCAL style languages might have ordered things contrary to FORTRAN, ALGOL and others. The language designer must consider these trivial but important alternatives. The layout of programming entities is essential for effective communication. The dual roles of language syntax, and programming style affect comprehension. A dictionary or index style layout suggests placing entity names first, followed by their definition.

3.24. Garbage Collection

One of the hallmarks of high level languages is that programmers declare data without regard to how the data is allocated in memory. In block structured languages, local variables are allocated on the stack, and automatically deallocated when the block exits. This relieves the programmer of a great burden. Garbage collection provides equivalent relief in languages with dynamic entity allocation.

In C++ the programmer must manually manage storage due to the lack of garbage collection. This is a difficult bookkeeping task that leads to two opposite problems. Firstly, an object can be deallocated prematurely, while valid references still exist (dangling pointers). Secondly, dead objects might not be deallocated leading to memory filling up with dead objects (memory leaks). Attempts to correct either problem can lead to overcompensation and the other problem occurring. A correct system is a fine balance. This is illustrated in the figure below.



These problems contribute to the fragility of C++ programs, and usually result in system failure. Garbage-collection solves both problems. Garbage-collection has an undeserved bad reputation due to some early garbage-collectors having performance problems, instead of working transparently in the background, as they can and should. These problems are often over-emphasised as a justification for C++ ignoring garbage collection. A possible solution is to build garbage collection into the run time architecture, but allow the programmer to activate and deactivate it manually. Garbage collection can be disabled in systems where it is inappropriate.

In C++ it might be argued that the lack of garbage-collection is not an engineering compromise. Its inclusion is nearly an engineering impossibility, as a programmer can undermine the structures required for implementing correctly working garbage-collection. While garbage-collection might not actually be an impossibility in C++ (EC++), it is difficult, and programmers would have to settle for a more restricted way of programming. This could be a good thing. But then the compromise to remain compatible with C becomes difficult, if the compiler is to detect practices inconsistent with the operation of garbage-collection.

3.25. 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. 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 ir-responsible.

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 does the C++ compiler (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? There is no reason why C++ should not be implemented this way. 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 manner of object-oriented reusability.

In the future, Unix could 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 developers. But this is the sort of software now being demanded by end users. Systems that are cobbled together are unacceptable.

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’.

3.26. C++ and the software lifecycle

The software lifecycle has attracted a great deal of attention. It is at least generally accepted that the activities in the lifecycle are analysis of requirements, design, implementation, testing and error correction, extension. Unfortunately, the result of identifying these activities has resulted in a school of thought that the boundaries between these activities are fixed, and that they should be systematically separate, each being completed before the next is commenced. It is often argued that if they are not cleanly separated, then you are not practicing disciplined system development.

This view is incorrect. Someone who writes a program straight away is actually doing all the steps in parallel. It might not be the best way do do things in many circumstances, might or might not suit the style and thinking of different people, but this works in some scenarios, and can be the methodology of choice of disciplined thinkers. Some people can hold a whole problem and solution in their head and work in a disciplined fashion until the solution is complete. Mozart is said to have composed this way, producing his last three symphonies in as many months in 1788. Beethoven toiled far more over the production of his works, taking years to complete one symphony. Both composers produced masterpieces. Mozart wrote music directly, whereas Beethoven wrote themes and ideas in his famous sketchbooks. The production of masterpieces depends on skill, not on methodologies.

It is becoming accepted that the software lifecycle should be an integrated process. Analysis, design and implementation should be a seamless continuum.The activities of the lifecycle should progress in parallel to expedite software development. Facts found out only as late as the implementation stage can be fed back into the analysis and design stages. The object-oriented approach supports this process. Artificial separation of the steps leads to a large semantic gap between the steps. The transformations required to bridge such semantic gaps are prone to misinterpretation, time consuming and costly.

The same people should be responsible for all stages. This way they take responsibility for the system as a whole, rather than passing the buck and blame which occurs when analysts, designers and implementers are different groups. This is not a popular viewpoint in traditional hierarchical management structures where programmers get promoted to designers who get promoted to analysts. Hierarchical management also discourages people from feeling responsible for a product. This culture must radically change if we are to produce quality systems.

We should have learnt from the extremes SA/SD. Some quarters believed that methodology was all important, while programming and programming languages were unimportant. Arcane and machine-oriented programming languages strengthened this attitude. These languages concentrate on the ‘how’ of computation, whereas the modellers correctly demand notations that express the ‘what’, in order to be implementation independent. A modern software language supports the integration of the activities of design and implementation by being readable, and problem-oriented. A language should be as close to design as possible. The needs and requirements of an enterprise can change much more rapidly than programmers can keep up, especially in a highly competitive and commercial world.

So how does C++ fit into this picture? Well it is based on C that was designed mainly as an implementation and machine-oriented language. It is an old language, that did not need to consider the integrated lifecycle approach. C++ might have

some of the trappings of object-oriented concepts, but it is the marriage of a problem-oriented technique with a machine-oriented language. It addresses implementation, but not so well the other aspects of the software lifecycle. Since C++ is not so well integrated with analysis and design, the transformation required to go from analysis and design to implementation is costly. The semantic gap between design languages and the implementation language is great.

We should have learnt from the structured world that this is the incorrect approach to the software lifecycle. But in the OO world we are again falling into the trap of dividing the lifecycle into artificially distinct activities of OOA, OOD and OOP, instead of adopting an integrated approach to these. Modern languages provide a much more integrated approach to the complete software development process than C++. C++ supports classes and inheritance and other concepts of object-orientation, but fails to address the entire software lifecycle.

3.27. Reusability and Communication

Reusability is a matter of communication.

In order to use a software component, you must be able to understand it. The writer must communicate the purpose, intent, and correct usage of the component to the client. In the object-oriented world, clear and concise definition of software modules is not a mere nicety, but essential for reusability. Arising out of the issue of reusability is extendibility. In order to maximise the reuse of software, it often must be tailored for new applications. The client programmer must decide whether the software component is suitable for the new task. If so, what is the best way to extend it? Clear communication to clients is a courtesy concern.

3.28. Reusability and Trust

Reusability is a matter of trust.

Trust results from confidence that safety concerns have been met. If you do not have confidence in a software component, then it is difficult to consider it for reuse. There could be doubt that the software component provides enough functionality, or correct functionality. There could be doubt that the component is efficient enough, or worse it might crash. The C/C++ philosophy of not building checks into the language and compiler because programmers can be trusted, works against trust and reusability.

In the real world of reusability, the ideal of trusting programmers is inappropriate. Trusting programmers results in less trustworthy software. In reality, customers doubt the claims of suppliers. It is the onus of the supplier to prove their claims, and thus trustworthiness of the software. The client is not required to trust the supplier's programmers. Potential clients of a software component, require assurance that the component is trustworthy. Trusting programmers is against the commercial interest of both parties. This is not to cast dispersion on programmers, but merely recognises that computers are good at performing mundane tasks and checks, but people are not. If people were good at such things, we would not need computers in the first place. Building trustworthy components is a safety concern.

3.29. Reusability and Compatibility

Different compiler implementations need to be compatible in order to realise reusability between components. Different C++ compilers generate different class layouts, virtual function calling techniques, etc. The name encoding schemes used for type safe linkage can also be different. If two different compilers generate different run-time organisations, then different name encodings are desirable as it will prevent two incompatible libraries from being linked. The C++ ARM (p122) states "If two C++ implementations for the same system use different calling sequences or in other ways are not link compatible it would be unwise to use identical encodings of type signatures."

This can be solved in two ways. Firstly, a library vendor could provide the entire source of a library so it can be compiled by the customers compiler. This is not satisfactory if the sources are proprietary. Then the vendor will need a separate release for every environment, and every compiler in that environment.

Because of this problem a strong case exists for a universal intermediate machine readable representation of programs. Interestingly, some systems are already using C as a 'universal assembler', notably AT&T C++ and Eiffel. But this cannot solve the above problems of compatibility between components without a standardisation effort on run time layouts and name encoding schemes.

3.30. Reusability and Portability

Since true OOP ensures that objects are loosely coupled to the external environment, portability to diverse environments is possible. C is tightly coupled to the Unix environment, and as such is not particularly portable to diverse environments.

3.31. Idiomatic Programming

The ability to program in different idioms is argued as a strength of C++. Idiomatic programming, however, is a weak form of paradigmatic programming. It is programming in a paradigm without necessarily having compiler support for that paradigm. The compiler cannot check for inconsistencies with the idiom, or paradigm. Defines can often be used to invent idioms. Anyone who has attempted to do object-

oriented programming in a conventional language using defines will realise that it is impossible to realise all the benefits easily, if at all, without compiler support.

3.32. Concurrent Programming

In the next ten years multiple processor arrays that execute programs concurrently will probably become common. Concurrency requires much cleaner languages, than the single processor languages of today. Object-oriented concepts support concurrent programming. Objects can execute state changing code independently of each other. Concurrent programming will be enabled by the division of the state space of a system into modules to achieve a high degree of independent processing. Objects provide a scheme to cleanly divide state spaces. The demand that everything be broken down into loosely coupled modules, that only interact through well defined interfaces might be perceived as inefficient. But it is precisely this scheme that will mean that concurrent solutions can be developed efficiently and transparently to the programmer. Concurrency should be transparent to the programmer, as concurrency is a low level implementation consideration. That is concurrency is how a computation is done, not what is to be computed. The programmer should be concerned with what is to be computed, not how. How something is computed is the concern of the target environment, ie the compilers, operating system, and hardware. When programmers are not concerned with this level, efficiency and portability follow automatically.

The aim of concurrent processing is to keep all the processors in a processor array as fully utilised as possible, so that processor resources are not wasted. This is as good as can be expected. There is nothing more mysterious to concurrent programming than the efficient use of resources. Keeping all processors busy is an inherently dynamic problem, which the programmer cannot determine statically at compile time. All the processors can be kept busy, as long as there are enough threads in the system.

In concurrent programming, a thread is a unit of sequential execution. Concurrency is achieved by the splitting of threads. A thread can be split when a state changing routine is invoked, but not a value returning function, because it must wait for the value. State changing routines can easily be invoked on another processor. Object level granularity seems to be a natural candidate for concurrent processing. An object can have only one update thread at a time to avoid simultaneous update problems. Other levels of concurrency are instruction level, and task or process level. Task or process level is the level used in conventional multi-processing systems currently commercially produced, and instruction level is quite difficult, best being left to instruction pipelines.

Object level is natural for the programmer, and has the advantage that a programmer can implement a system without taking into account parallel processing at all. The same program will run and produce identical results irrespective of whether the customer is running a single processor, or a processor array.

Side effects must be avoided in concurrent systems. Suppose a computation depends on combining the results of two functions f and g, such as f + g. If f and g are independent, then they can be computed concurrently. If however, f produces side effects that g depends on, they must be computed sequentially. F and g are parameters to the + function. Routine parameters can be computed concurrently, as long as the computation of each causes no side effects. Side effects are avoided by restrictive practices that C devotees would object to.

C++ does not preclude the use of a global environment. Access to shared global data potentially causes a thread to lock, and if many such accesses occur, the advantage of concurrency is lost. This is because updates to a global environment are side effects. Programming in such an environment requires complex locking mechanisms to ensure that things happen in the correct order. Locks are rather like waiting for a plane to take off when it has to wait for another connecting flight. This cannot be entirely avoided, but should be reduced as much as possible.

4. The role of Language

For an intermission between sections, I'll mention some interesting points that the Cambridge Encyclopedia of Language [Crystal 87] makes. It says that language is an emotional subject. "It is not easy to be systematic and objective about language study. Popular linguistic debate regularly deteriorates into invective and polemic. Language belongs to everyone; so most people feel they have a right to hold an opinion about it. And when opinions differ, emotions can run high. Arguments can flare over minor points of usage as over major policies of linguistic planning and education."

While natural language is difficult to be "systematic and objective about", should this apply to computer languages? The definition of natural language is generally beyond our control, with the exception of languages such as Esperanto. Programming language definition, however, is within our control. Programming languages must have expressiveness like natural language, yet be precise and semantically consistent. As programming languages have rigorous requirements, we should be even more critical and objective about them. It is a measure of immaturity in the programming profession that emotional and irrational defensiveness often denies valid criticism. Many dismiss the choice of

programming language as a religious issue. If language choice is merely religious, then we might as well still program in assembler, or maybe even binary, because the adoption of high level languages would have no technical merit. Language choice, however, is a technical consideration. Technical measures should judge the effectiveness of a language. Understanding the role of language helps quantify what must be measured.

The Cambridge encyclopedia lists several functions of language. “To communicate our ideas”, it says is the most common answer, and this must surely be the most widely recognised function of language. It lists several other functions of language. One function is emotional expression. For instance, when we stub our toe, we often emit words, even when there is no one to hear. Another is social interaction. For example if someone sneezes, we often “bless” them. Another is the power of sound, as in poetry and rhyming jingles etc. Another is the control of reality, as in spells and incantations. Perhaps computer programs and spells are similar in purpose. Another is recording facts. This includes record keeping, historical and geographical documents, etc. Another is the instrument of thought. We often reason about things to ourselves in language. Another function is the expression of identity. Language can express who we are, or affirm our belonging to certain groups. Perhaps the most important role of computer languages is to enable description, and recording the decisions made during the design and implementation of a system.

Since language and communication are two closely related concepts, it is important to understand their relationship, and the nature of communication. Language is the set of aural and written symbols with which we communicate. Laurence Wylie in the foreword to “French in Action” [Capretz 87] describes communication as “To understand this [communication] we must know the basic meaning of the words *common*, *communicate*, and *communication*. They are derived from two Indo-European stems that mean “to bind together.” In this ordered universe, no human being can live in isolation. We must be bound together in order to participate in an organised effort to accomplish the necessary activities of existence. This relationship is so vital to us that we must constantly be reassured of it. We test this connection each time we have contact with each other.”

The concept of binding is also important in computing. In networks, binding establishes communication links between two or more entities. This forms a greeting, so that a relationship is established, and communication is possible. In programming we have the concepts of static and dynamic binding. Binding in this paradigm makes it possible for a message to be sent from one object to another, so that they can communicate and interact. Static binding determines this message in advance as the receptor is always the same type, or descendant of that type. Static typing ensures in advance that the receptor object can process the message. Dynamic binding, means that the exact message to be sent is determined by the dynamic type of the receptor when the message is actually sent. For example, on your telephone, you talk to your friends differently than a client, even though you are using the same piece of equipment. These are the concepts that C++’s virtual do not express well. Designing an object-oriented system is like designing a language by which objects interact. Thus tools used for formal programming language design, BNF, denotational semantics, and axiomatic semantics can help in the design of an object-oriented system.

“Language shapes the way we think, and determines what we can think about.” - B.L.Whorf. Bjarne Stroustrup quotes this in “The C++ Programming Language”. But is this correct? The encyclopedia says, “It seems evident that there is the closest relationship between language and thought: everyday experience suggests that much of our thinking is facilitated by language. But is there identity between the two? Is it possible to think without language? Or does language dictate the ways in which we are able to think? Such matters have exercised generations of philosophers, psychologists, and linguists, who have uncovered layers of complexity in these straightforward questions. A simple answer is certainly not possible; but at least we can be clear about the main factors which give rise to complications.”

The above Whorf quote is a statement of the Sapir-Whorf hypothesis on language and thought. Edward Sapir (1884-1939) formulated this with his pupil Benjamin Lee Whorf (1897-1941). It reflects the view of its day when great value was placed on the diversity of the languages and cultures of the world. The Sapir-Whorf hypothesis combined two principles. The first is ‘linguistic determinism’, which states that language determines the way we think. The second, ‘linguistic relativity’, that the distinctions found in one language are not found in any other. There can be both verbal and non-verbal thought; following a road map in a car for example. Street directions are often difficult to put into words.

The Sapir-Whorf hypothesis in its strongest form, as in the Whorf quote, is now not generally accepted. For one reason, it is known that concepts can be translated from one language into another. This is even if in one language, the concept can be expressed in one word, but takes a phrase of words in another. A weaker version of the Sapir-Whorf hypothesis is accepted. That is “language may not determine the way we think, but it does influence the way we perceive and

remember, and it affects the ease with which we perform mental tasks. Several experiments have shown that people recall things more easily if the things correspond to readily available words and phrases. And people certainly find it easier to make a conceptual distinction if it neatly corresponds to words available in their language. Some salvation for the Sapir-Whorf hypothesis can therefore be found in these studies, which are being carried out within the developing field of psycholinguistics."

The important question to the programming community is do programming languages 'shape' the way we think about and design systems? The negative argument is that it is the concepts behind languages that are important, not the languages themselves. Languages only provide a framework for the expression of the concepts. A language can only be as good as the concepts it implements. A programming language influences the way we program, and the way we use the concepts it implements. It can clarify the concepts, or obscure them as in the case of C++. A language must implement the concepts cleanly and simply. It must express the concepts in as few words and constructs as possible. But this does not just mean avoiding keywords as in C. Programmers who understand the concepts should have no difficulty in adapting to different languages, as long as the new language implements the concepts elegantly.

A language can be judged like a wine connoisseur judges wine, by holding it up to the light to judge for clarity and colour. Ultimately, it is the taste that matters, but good colour and clarity suggests that the taste is more likely to be good. Clear programming language definition helps in the goal of the production of quality software.

So where does this leave Sapir-Whorf with respect to programming languages? Programming languages do not shape the way we think. It is the concepts that shape the languages, and it is the way we think that shapes the concepts. Those who have attempted to learn a language in order to learn object-oriented programming realise that it is the concepts which must be grasped in order to be effective. Once the concepts have been learnt, object-oriented programming seems a natural way to program. It matches very effectively the way we think. If C++ has been designed according to the Sapir-Whorf hypothesis, its philosophical basis does not serve a computer industry that should shape tools best suited to its purposes, processes, thinking, and concepts.

5. On Writing

During the development of this critique, I realised it had grown larger than I had intended, and that my writing style needed some polish for such a large work. During my research, some colleagues recommended a small book, "The Elements of Style," [S & W 79]. This has been around for most of this century in one form or another. William Strunk, the original author had some stern advice for his students:

"Vigorous writing is concise. A sentence should contain no unnecessary words, a paragraph no unnecessary sentences, for the same reason that a drawing should have no unnecessary lines and a machine no unnecessary parts."

The machines that the software professional develops are not built, but written. Strunk's last sentence prompts consideration of the relationship of writing to software development. A common situation is taking several thousand lines of incomprehensible 'code', and making it execute efficiently. After spending considerable time we often realise what the program does, and reduce it to several hundred lines of program 'text' that runs ten times faster. Strunk's quote should be applied to programming. A routine should contain no unnecessary declarations or instructions, a system no unnecessary routines. It can also be applied to programming languages. A programming language should contain no unnecessary constructs. This is the root of my dissatisfaction with C++. Much of it is unnecessary, even for the most complex systems. Its syntax is ugly. C++ has become what the C world has constantly criticised in languages like PL/1 and Ada. Only C++ is worse. We need to regain artistic elegance and simplicity.

6. Generic C criticisms

These criticisms apply to the C base language, but in general adversely affect C++. R.P.Mody [Mody 91] gives an excellent general criticism of C. He says that to properly understand C you must understand the insides of the compiler. He gives many examples of how C obscures rather than clarifies software engineering. He concludes that he is "appalled at the monstrous messes that computer scientists can produce under the name of 'improvements'. It is to efforts such as C++ that I here refer. These artifacts are filled with frills and features but lack coherence, simplicity, understandability and implementability. If computer scientists could see that art is at the root of the best science, such ugly creatures could never take birth."

C's popularity is based on several myths. Firstly, that it is a high level language. It is not. It is a structured assembler oriented towards the low level machine domain, not to the problem domain of a high level language. Secondly, that it is small and simple. Its semantics are not simple, but it is very simple to make catastrophic errors. Thirdly, that it is portable. Certainly compilers are available on many platforms, but this does not make programs portable, especially to diverse, and future architectures. Platform independence achieves portability. Fourthly, that it is efficient.

What seems efficient on some platforms is the very antithesis of efficiency on other platforms. It seems efficient on certain platforms because it allows the lower level machine-oriented architecture to be visible at a higher level, instead of being handled transparently by the compiler. This means that programs will be locked into certain styles of architecture, or into current styles of technology, instead of protecting program investment against future technological change. And lastly, that the semantics are mathematically rigorous. Anyone who reads the C++ ARM will realise just how poorly defined the language is. Anyone who has practiced C will know how many traps there are to fall into.

6.1. Pointers

C pointers are a low level mechanism that should not be the concern of programmers. Pointers mean the programmer must manipulate low level address mechanisms, and be concerned with lvalue and rvalue semantics, which are machine oriented and not problem oriented as you would expect of a high level language. A compiler can easily handle such issues without loss of generality or efficiency. Memory models of different environments often affect the definition of pointers. Memory model details such as near and far pointers should be transparent to the programmer.

The programmer must also be concerned with correct dereferencing of pointers to access referenced entities. Use of pointers to emulate by reference function parameters are an example. The programmer has to worry about the correct use of &s and *s. (See the section on function parameters.)

Pointer arithmetic is error prone. Pointers can be incremented past the end of the entities they reference, with subsequent updates possibly corrupting other entities. How many lurking and undetected errors are in programs because of this? This illustrates how C undermines OOP by providing a mechanism where state outside an object's boundaries can be changed. Since pointers are intrinsic to writing software in C this exacerbates the problem. Pointers as implemented in C make the introduction of advanced concepts like garbage collection and concurrency difficult.

Another consideration is that dynamic memory implementations vary between platforms. Some environments make memory block relocation easier by having all pointers reference objects via a master pointer which contains the actual address of the block. The location of the master pointer never changes, so relocation of the block is hidden from all pointers that reference it. When the block is relocated, only the master pointer needs to be updated.

On the Macintosh, for example, the double indirection mechanism of 'handles' facilitates relocation of objects. Object Pascal makes these handles transparent to the programmer. This is similar to the Unisys A Series approach where object 'descriptors' access the target object via a master descriptor that stores the actual address of the object. On the A Series this is transparent to programmers in all languages, as this transparency is realised at a level lower than languages. The A series descriptor mechanism also provides hardware safety checks that mean that pointers cannot overrun, and arrays cannot be indexed out of bounds. C cannot be implemented particularly well on such machines, as C's mechanisms are lower level than the target environment.

Other environments do not provide object relocation, so double indirection is an unnecessary overhead. In order for programs to be portable and to be at their most efficient in different target environments, such system details should be the concern of the target compilation system, not of the programmer.

C's pointer declaration syntax causes another small problem:

```
int*      i, j;
```

This does not mean, as might be easily read -

```
int    *i, *j;
```

but

```
int    *i, j;
```

and should be written thus to avoid confusion.

6.2. Arrays

Page 137 of the C++ ARM notes that C arrays are low level, yet not very general, and unsafe. Page 212 admits, "the C array concept is weak and beyond repair." Modern software production is far less dependent on arrays than in the past, especially in the object-oriented environment. The trade off to be optimal, rather than general and safe no longer applies for most applications. C arrays provide no run-time bounds checking, not even in test versions of software. This compromises safety and undermines the semantics of an array declaration, ie an array is declared to be a particular size, and can only be indexed by values within the given bounds. An index to an array is a parameter in the domain of the array function. An index out of bounds is not a member of the domain, and should be treated as severely as divide by zero. C has no notion of dynamically allocated arrays, whose bounds are determined at run time, as in ALGOL 60. This limits the flexibility of arrays. The C definition of arrays compromises both safety and flexibility.

One view of arrays is just another object-oriented entity which should be treated in an object-oriented manner as a class of data structure. It should have interface definitions, and consistency checks inherent in object-oriented systems. Another view is that an array is an

implementation of a function, where pairs of values explicitly map the domain to the range, rather than being computed. This suggests that Algol was incorrect in distinguishing arrays by using square brackets. An array just maps the input argument (the index) to a value of the type of the array. An array can be viewed as a random access stack.

[Ince 92] considers that arrays and pointers need not be relied upon so heavily in modern software production, as higher level abstractions such as sets, sequences, etc are better suited to the problem domain. Arrays, and pointers can be provided in an object-oriented framework, and used as low level implementation techniques for the higher level data abstractions. As has already been mentioned object-oriented programming is very useful for the encapsulation of implementation and environment oriented details. Ince suggests that arrays and pointers should be regarded in the same way as gotos in the seventies. He suggests that languages such as Pascal and Modula-2 should be regarded in the same way as assembler languages in the seventies. This applies even more to C and C++, because pointers and arrays are far more intrinsic in the use of C and C++.

I agree with Ince that we have less dependence on arrays, and that pointers in programming languages can be considered harmful. But I disagree in as far as the concept of array is useful for mapping one set of values onto another, where this mapping cannot be described computationally, but can only be expressed by pairs or tuples of coordinates.

6.3. Function Parameters

Parameters are used to pass routines simple values (by-value parameters), or references to entities (by-reference parameters). Parameters are inputs to routines, and should not be changed. When memory was expensive, reusing parameter space could conserve space. Changing parameters, however, is semantic nonsense, and most languages get this wrong.

By reference parameters enable a routine to change the value of an entity external to the routine. Such updates beyond the environment of a routine are side-effects. This introduces a mechanism of updating the state space, other than straight assignment (although the routine can use assignment to achieve the ‘dirty deed’.) The danger is that the state of an object can be changed without using the well defined interface of the object. By-reference parameters should not be used to change the external world. Values should only be passed to the external world by the return value of a function. Semantically, this is quite different to assignment to a reference parameter; data flows through the program in one direction, in via parameters, and out via return values. Mathematically this maps compositions of inputs to outputs. Abstract data types can be used to design such systems. Also this will help target environments to increase parallelism and concurrency in a way transparent to programmers.

In object-oriented programming, by reference parameters are used to pass the original object, not a copy. The called routine, however, cannot change the state of the referenced object. Only calling a routine in the objects interface can change the state. This has the desired effect of the object being given to you, without being yours to change, although you can effect change in the object.

C shares faulty parameters with many other languages. The interaction of C’s pointer mechanism with a faulty parameter mechanism, however, makes C considerably worse than most other languages. In C, pointers are used to simulate by-reference parameters with by-value parameters. The programmer must perform tedious bookkeeping by specifying *s and &s for referencing and dereferencing. Distinguishing between by-value and by-reference parameters is not just a syntactic nicety, included in most high level languages, but a valuable compiler technique, as the compiler can automatically generate the referencing and dereferencing, without burdening the programmer.

6.4. void *

“Passing paths that climb half way into the void” - Close to the Edge, Yes.

Is void * the C equivalent of an oxymoron? A pointer to void suggests some sort of semantic nonsense, a dangling pointer perhaps? Maybe we should tell the astronomers we have found a black hole! While we can have some fun conjecturing what some of the obscure syntax of C suggests, a serious problem is that void * declarations are used to defeat the type system, and so compromise its purpose. A well thought out type system does not require such a facility. In an object-oriented type system, the root class of the inheritance hierarchy provides the equivalent of void.

When a typed entity is assigned to a reference of void *, it looses its static type information. When it is assigned back to a typed reference the programmer must explicitly specify to the compiler the type information. This is error prone and should at least result in a run-time check, to make sure that the correct type actually is being assigned. Without type checks, the routines of one class can be mistakenly applied to objects of another class.

6.5. void fn ()

The default type that a function returns is int. A typeless routine returning nothing should be the default. Instead this must be specified by another confusing use of void. This is an example of

where C's syntax is not well matched to the concepts and semantics. Syntactically no <type> should suggest nothing to return. Also a typed function can be invoked independently of an expression. This is a shorthand way of discarding the returned value. Values should be returned because they need to communicate with the outside world, and ignoring returned values is often dangerous. In other words, using a typed function as a void should result in a type error.

In fact there should be no such thing as a void function. A void function is a procedure. Procedures and functions should be distinguished. This distinction belongs to the problem 'what' domain. A procedure is a routine that changes the state of its object, but returns no value. A function should, in general, not cause any change to the state of an object, but just return some result dependent upon the objects state. Mathematically, a function is an entity that returns a value of a given type. Procedures are untyped, and do not return a value. So it is incorrect to regard procedures as functions. Functions as will be explained below have more in common with variables than procedures. Procedures can cause side effects, functions should not cause side effects. These distinctions are useful when considering concurrency.

6.6. fn ()

Empty parenthesis represent the function invocation operator in C. Even though '()' is mathematical looking, it is semantically equivalent to FORTRAN's CALL, COBOL's PERFORM, and JSR in assembler. The design of these operators was influenced by the underlying machine architectures. The invocation operator is low level, machine and execution oriented, and in the 'how' domain.

This is opposite to most Unix shells, where invocation operators such as 'run' and 'exec' are not needed. The ability to execute file names as commands extends the command repertoire. The shell runs executables and interprets shell scripts. There is no distinction as far as the shell user is concerned. This is a widely accepted as an elegant and effective convenience. C's () operator introduces the equivalent of a run command into the language.

No invocation operator exists in the problem oriented domain of high level languages. This is because the semantics of a function is to return a value of a given type. How this value is computed is unimportant. The value could be computed by a routine invocation, by sending a message across a network, by forking an asynchronous process, or by retrieving a precomputed result from a memory location, ie a variable.

The distinction that languages like C make between variables and value returning routines is artificial. It could be pointed out that variables are fundamentally different to functions, as you can assign values to variables. Functions, however, are the target of assignment. The return statement accomplishes this in C. Algol has no return statement, but uses assignment to the name of the function. The assignment of a value to a variable sets the return value for subsequent invocations of that function.

It is trivial for a compiler to realise this transparency of view for variables and functions. In ALGOL style languages, the compiler automatically deduces invocation when it sees a name that was declared as a routine, rather than a variable. The compiler knows that the identifier refers to a routine. This compiler technology was not realised when FORTRAN and COBOL were developed. This compiler technology is possible, because the compiler stores much information about an entity. A compiler can check that the programmer uses the entity consistently with the declaration. A compiler can generate correct code, without burdening the programmer with having to redundantly use an invocation operator. This enhances flexibility and implementation independence.

In fact the Unisys A series architecture elegantly achieves this level of transparency at the hardware level. The value call (VALC) operator loads a value onto the top of stack. When VALC hits a data value, that value is retrieved from memory and loaded onto the stack. When VALC hits a program control word (PCW), a routine that computes the value to be loaded onto the stack is invoked.

Variables and functions should be interchangeable for programmer optimisation. In C, it is not possible to change a function to a variable without removing all the (). This might be spread over many files, and the programmer might not bother with optimisation to avoid the tedium of the task. So the () operator reduces flexibility. Thus implementation detail is visible for the outside world to see. The () operator is another bookkeeping task imposed on the C programmer. Pure functional languages such as SML remove the variable/function distinction altogether, by not having variables at all.

The removal of the variable/function distinction would remove the need for a common use of C++'s inline functions. Inlines clutter the name space of a class and add work for the programmer. All that is required is to directly export a data member as a function.

C also has pointers to functions. Function pointers are analogous to the call by name facility in ALGOL, and this was recognised as having pitfalls. Consistent application of the object-oriented paradigm avoids these pitfalls. A common use of function pointers is to explicitly set up jump tables. The mechanism behind virtual functions is a jump table of function pointers. The design of a program can take advantage of this fact, without resorting to explicit jump tables.

Another use is to jump to a function in a table that is indexed by an input character. A switch statement can cater for this mechanism that makes what is meant explicit, while keeping underlying mechanisms (and possibly optimisations) transparent. C++ allows function pointers to member functions to be stored in tables (via the .* and ->* operators).

6.7. Metadata in Strings

The implementation of strings in C mixes metadata with data. Metadata is data about an object, but is not part of the data itself. Examples of metadata are addresses, size and type information. Such metadata is often referred to as data descriptors, and can be kept independently of the data, with the advantage that the programmer cannot mistakenly corrupt the metadata.

In C strings, metadata about where a string terminates is stored in the data as a terminating byte. This means that the distinction between data and metadata is lost. The value chosen as the terminator cannot occur in the data itself. The common alternative implementation is to store a length byte in a fixed location preceding the string. This length metadata can be hidden from the programmer who does not need to know where the length metadata is stored. This implementation also has the advantage that the length of a string can be easily obtained, without having to count the number of elements up to the terminating null.

6.8. ++, --

The increment and decrement operators are often used as an example that C was designed as a high level assembler for PDP machines. These operators provide a shorthand convenience, but are unnecessary. There are no less than three ways to perform the same thing -

```
a = a + 1
a += 1
a++
++a
```

For full generality, only the first form is required, the others are a mere convenience. The last two forms a++ and ++a are the postfix and prefix forms. They are often used in the context of another expression. Thus several updates can be performed in one expression. This is a very powerful and convenient feature, but introduces side effects into an expression that sometimes have surprising effects, and can lead to program errors. The following example is given on p.46 of the C++ ARM -

```
i = v[i++]; // the value of 'i' is
            // undefined
```

The ARM points out that compilers should detect such cases, but the exact interpretation appears to be left to the implementation, which contributes to non-portability. If this can't be defined for a sequential processor, then it is even worse for a concurrent environment.

The shorthand += and -= are more powerful as values other than 1 can increment the variable. It has been suggested that there should also be &&= and ||= operators.

If it is mistakenly believed that a multiplicity of operators is required to produce more optimal code, then it should be pointed out that code generators, especially for expressions, can produce the best code for a target architecture. A plethora of operators complicates the task of an optimiser. A compiler can optimise well beyond what a programmer can do. An optimising compiler will analyse the surrounding code, and if an entity is used several times in a local scope, it will keep the value of that entity handy locally at the top of a stack, or in a register, rather than retrieve it from slow main memory several times. The nature of such optimisations depends on the machines architecture, which a programmer should not have to be aware of. Open systems demands that programs can be ported amongst diverse architectures and environments, very different to the original machine, and not only run, but run efficiently. Optimisers work best with simple, well defined languages.

In fact constructs such as:

```
while (*s1++ = *s2++);
```

might look optimal to C programmers, but are the antithesis of efficiency. Such constructs preclude compiler optimisation for processors with specific string handling instructions. A simple assignment is better for strings, as it will allow the compiler to generate optimal code for different target platforms. String assignment will also hide the implementation details of strings. If the target processor does not have string instructions, then the compiler should be responsible for generating the above loop code, rather than requiring the programmer to write such low level constructs. The above loop construct for string copying is also the contrary to safety, as there is no check that the destination does not overflow. The above code also makes explicit the underlying C implementation of strings, that are null terminated. Such examples show why C cannot be regarded as a high level language, but rather as a high level assembler.

As with name overloading, memory storage update is a problematic, but necessary part of programming. A language should provide it in a consistent and expected way. Many languages recognise that memory update is problematic, and typically only provide limited but sufficient ways of updating, by an assignment operation. (Many languages have block memory copies as well, but assignment can also provide block copy.) Furthermore, many languages avoid side-effects

by limiting updates to only one per statement. C provides too many ways to update memory. These add nothing to the generality of the language, increase the opportunity for error, and complicate automatic optimisation. Restrictive practices are justifiable in order to accomplish correctly functioning and efficient software.

6.9. Defines

The define declaration -

```
#define d(<parameters>)
```

has a different effect to -

```
#define d  (<parameters>)
```

The second form defines d as (<parameters>). Extra white space between tokens should not affect semantics of constructs.

#defines are poorly integrated with the language. The '#define' must be in column 1, and knows nothing about scope rules. Errors in defines can lead to obscure errors, as the preprocessor does not detect them, but leaves them for the compiler. Programmers must be familiar with the particular preprocessor implementation on their system, as preprocessor implementations are different, particularly between Classic C and ANSI C.

6.10. NULL vs 0

[Ellemtel 92] recommends that pointers should not be compared to, or assigned to NULL, but to 0. Stylistically, NULL would be preferable. It would also allow for environments where null pointers have a value other than 0. ANSI-C, however, has subtle problems with the definition of NULL.

6.11. Case Distinction

It is good to adopt typographic conventions for names, but distinguishing between upper and lower case in names can cause confusion. Confusion leads to errors and systems that are difficult to maintain and modify. Case distinction is based on the implementation paradigm of how character codes work. Why do we have names? To give entities identity, and aid our memory of that identity. Philosophically, case distinction is contrary to the fundamental purpose of names.

Case distinction in interactive systems is a poor user interface. It is clumsy having to continually use the shift key, and will slow a good typist. More importantly, case distinction makes names harder to remember, and so is contrary to the purpose of aiding memory. It is difficult enough for users to remember command mnemonics or file names, let alone exactly the case. Names are used instead of difficult to remember addresses. If we did not have names, we would have to retrieve files by addresses, or call people by their social security number.

Consider the paradigm of letters and words. Words are spelt by assembling letters in order. There are 26 distinct letters. With the addition of digits 0 to 9, and the underscore character, we have a complete and correct definition for identifiers. Letters can be written in a number of styles. They can be bold, italic, upper or lower case. Such typographic representations, however, do not change the semantics of a word. Thus if we write ALGOL, Algol or algol, we recognise the word to represent a computer language. The case of the letters does not change the semantics. Letter case is only a typographic device. Typographic conventions make program text more readable, but should not affect the semantics of a program. Case distinction is based on the low level paradigm of character codes such as ASCII used internally in the computer. This weakens the purpose of using names to replace addresses, as names are reduced to a string of character codes.

Case distinction also contributes to errors. It introduces ambiguity, and as has already been mentioned, ambiguity weakens the purpose of names, as identity is lost. As every programmer will have experienced, one character errors are more difficult to find than one would think. For example if an identifier is declared Fred, another one can be declared fred. Such names are easily mistyped and confused. We are in general poor proof-readers. The psychological reason for this is that the the brain tends to straighten out errors for our perception automatically. The human brain is an excellent instrument for working out what was intended, even in the presence of radical error. (This makes us good at difficult tasks like speech recognition.) In order to overcome this programmers must use their powers of concentration to override this natural tendency of the brain. Distinguishing upper from lower case in names only adds another level of difficulty. Good language design takes into account such psychological considerations in these small but important details, being designed towards the way humans work, not computers. Such considerations of cognitive science make a big difference to the effectiveness of people, but do not have any impact at all on the efficiency of code generated for the computer. What is more important, people or computers?

Case distinction provides another form of name overloading. Name overloading is a double-edged sword. It leads to ambiguity, confusion and error. Name overloading as has been suggested in the section on name overloading should only be provided in controlled and expected ways, where overloading provides a useful function such as module independence or polymorphism. Where a name is overloaded in the same scope the compiler should report an error.

As another example, a commonly used technique is -

```
class obj
{
   int    Entry;

   void   set_entry (int entry)
   {
          entry = Entry;
   }
}
```

If you have not spotted the error in the above example, what was it supposed to mean?

6.12. Assignment Operator

Using the mathematical equality symbol for the assignment operator is a poor choice of symbols. Programming assignment is not equal to mathematical equality (:= != =). Language designers of ALGOL style languages realised they were semantically quite different, so took the care to distinguish, only using '=' to mean equality in the sense of mathematical assertion. In C the lack of distinction leads to error. It is easy to use = (assignment) where == (equality) is intended, which looks reasonable, but results in errors that are difficult to detect.

This leads to a more general criticism of C, in that it has a pseudo mathematical appearance. Few people are proficient at interpreting mathematical theorems, most passing over such sections in text, making the assumption that the mathematics proves the surrounding text. The pseudo-mathematical nature of C has this bad attribute of mathematical notation. It is difficult to read, while lacking the semantic consistency and precision of mathematical notation. One of the keys of reusability is readability.

6.13. Type Casting

Type casting is just a mechanism to map values of one type onto values of another type. This means type casting is no more than a specific form of mathematical function. Type casting has been useful in computer systems. Often it is required to map one type onto another, where the bit representation of the value remains the same. Type casting is therefore a trick to optimise certain operations. Type casting provides no useful concept that general functions cannot implement. Furthermore, type casting undermines the purpose of strongly typed systems. In many languages, the type system has not been consistently defined, so programmers often feel that type casting is necessary.

Mathematically, all functions perform type casting. 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 the value 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 the value i.
```

The hardware of many machines cater for such basic data types as character and integer, and it is entirely possible that a compiler will generate code that is optimal for any target hardware architecture. So many languages have character and integer as built in types. The object-oriented paradigm, however, 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. Here though, the programmer might wish to specify the use of two type conversion functions to 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 suited to the target machine. Such inbuilt class definitions might be a part of the standard language definition.

6.14. Semicolons

I am not concerned whether the semicolon is defined as a terminator or separator. Arguments that languages that define the semicolon as terminator are superior to those that define it as separator are, however, baseless. The semicolon as separator is really quite logical. It is based on viewing the semicolon as a statement sequencing or concatenation operator. It is therefore a binary operator, requiring both a left and a right hand side. Some people claim to find this concept difficult to understand, but if we consider it in the context of a mathematical expression, it would be silly to expect that an addition be written as:

```
a + b +
```

Another way to look at a separator is to consider the structure of a program. A program is a list of elements. The executable part of a program is a list of sequentially executed instructions. Elements in a list must be separated, and the semicolon is one syntax to separate elements in a list. The semicolon is therefore part of the syntax of the list, not part of the syntax of the individual instructions. Languages such as FORTRAN separated instructions by requiring that they be placed on different lines or cards. If an instruction overflowed a line, a continuation character was required, like the backslash in C. Well defined languages do not require continuation characters, as line breaks are unimportant, and have no effect on semantics. Languages should have very regular grammars, so that the semicolon could be an entirely optional typographic separator.

In natural language both the comma and semicolon are separators, only the full stop is a terminator. If the comma was a terminator, function invocations would look like:

```
fn (a, b+c, d, e,);
```

It is often argued that the semicolon as separator leads to irregularities. C's handling of the grammar of semicolons, however, leads to an irregularity in if/else's:

```
if (condition)
    statement1; /* Semicolon
                   required */
else
    statement2;

if (condition)
{
   statement1;
}  /* Semicolon must be omitted */
else
   statement2;
```

This is an irregularity, as a parser will reduce both of the above to the grammatical form:

```
if (condition) statement
else statement
```

(In fact why do conditions in C if and while statements have to have parentheses around them?)

7. Conclusions

C++ is overly complex. C is widely recognised as being a simple language. But even this is doubtful, as it has many operators, and a difficult precedence system. Its pointer style of programming is difficult. Overall, C has many traps that lead to difficult to detect errors in software. Object-oriented languages should provide sophisticated concepts in the simplest possible framework. Where the framework is not simple, the concepts are obscured. OOP addresses many issues in order to facilitate the production of complex and sophisticated programs. Many of these issues are addressed in implicit and subtle ways, but are lost in C++. Subtle errors can be introduced into C++ software in many ways, and furthermore, the combination of these will cause even further problems. C++ has devices for petty convenience, while sacrificing major conveniences and long-term correctness and safety. C++ forces the programmer to perform many administrative bookkeeping tasks that a compiler can easily do.

It can be considered what application domain C++ is relevant for? The answer to this is that C++ might be used as a better C. But for what applications is C relevant? C is relevant for low level Unix systems programming. It is not a generally applicable language in view of its low level nature, and its flaws. C is not applicable for large scale production. Hence C++'s attempt to improve it. C++, however, has not solved C's flaws, as I once hoped it would, but painfully magnified them. Better languages exist for higher level functions such as communications and networks, scientific work, compilers, etc. I envisage that C has a place as a high level assembler that can be used to implement small pieces of code on suitable platforms, where

efficiency is of prime importance. Thus the use of C would be limited and well controlled, rather like small assembler routines are currently used in some systems for the same purpose. Indeed the move to C++ should only be considered in the case of upgrading a body of C programs for backwards compatibility. In the case of new projects alternatives to C and C++ should seriously be considered.

Programming is the orchestration of change within a large state space. Object-oriented techniques provide a method of simple division and management of such state spaces. Managing such state spaces requires the simplest techniques, in order to guard against detectable inconsistencies that lead to errors in executable systems. C and C++ do not implement the simple management of a large state space, and allow many potential errors to go undetected. The role of a language as a tool cannot seriously be regarded as some authoritarian that stops us doing what we want or need to do, as many languages with type safety and consistency checks are often viewed. Programming languages should embody the collective wisdom of common sense practices that have been learnt over many years, by common and painful experience. C++ lacks the implementation of much of this wisdom. [Sakkinen 92] observes that much of the C++ literature has few references to external work or research. It fails to draw on the insights and progress made by many researchers. This leads me to believe that C++ is parochial and removed from the many advances that will make production of systems easier and more cost effective.

It is better to detect and avoid errors than to fix them. The fixing of errors happens many times during the development process. This slows down the development process, and is therefore costly. Good programmers in this context (often called ‘gurus’), are those who recognise symptoms, and recommend fixes. Good programmers in the better sense (often called ‘impractical idealistic dreamers’) adopt better practices (programming languages being a subset of these), that avoid error in the first place.

C encourages gurus who spout false wisdom on obscure subjects. Writing programs in C is often called ‘coding’. Coding is writing obscure encryptions that will later have to be decoded, by none else than a guru! C also encourages programming by guesswork. C programmers often solve ‘bugs’ by adding extra ()s, *s and &s, without understanding the problem. People who attain proficiency at this guesswork, are known as, well you guessed it, gurus!!

The view that correctness checks are training wheels for students, which gurus don’t need must be dispelled. Many disciplines have techniques to ensure correctness. For example, the metronome in music is not just for students, but will help an advanced musician ensure that the tempo of a piece is correct, and since playing to a metronome is more difficult, will help sharpen the musicians performance of the piece. The musician does not just view the metronome as an aid for beginners, or as something that restricts him to a set beat, but as a tool that helps produce a polished and professional performance. C should not be seen as a language to which you graduate after you have learnt to program in languages with safety checks. In fact changing to C or C++ is a great step backwards. Languages with consistency and semantic checks are essential aids to the production of professional software.

This paper has shown many cases where C++ uses old C mechanisms to provide things that can and should be expressed consistently within the object-oriented paradigm. For example type casting. The move to pure object-oriented languages will facilitate more consistent programming and avoid many typical errors that occur in software production. C++ also makes distinctions that belong in the ‘how’ implementation domain. For example, ‘.’ vs ‘->, and variables and functions. These make bookkeeping work for programmers, which should be handled by a compiler. But then C++ fails to make distinctions that belong in the ‘what’ problem domain. For example, procedures vs functions. Making distinctions in the ‘how’ domain adds inconvenience to the language. Failing to make distinctions in the ‘what’ domain limits the power and expressiveness of the language. The amount of change required in C++ to address the issues raised in this paper is seen as largely insurmountable.

A programming language is just a tool, in the same way that an axe is a tool. If the axe is blunt when chopping down a tree, then procedures, processes and methodologies could be invented to make it as effective as possible. But that leaves the real problem unsolved; that the axe that does the real work is blunt. So it is with programming languages. To develop a system, it must be implemented, and a programming language is the tool to do the real work. If the language is blunt, then procedures, processes and methodologies might alleviate the situation, but they do not solve the problem. Once the axe is sharpened, then real progress is made, and the procedures, processes and methodologies also become more effective. A good axeman will have good axe wielding technique, but given a choice of axes will choose the sharpest implement. A poor axeman could be ineffective with even a sharp axe, but the axe maker will still strive to produce the sharpest axe for the good axeman. The argument that poor programmers will produce bad programs in any language so we shouldn’t bother with better languages is fallacious.

As mentioned in the introduction, both sides of the analysis/design vs implementation debate

need to compromise in order to bridge the semantic gap. The perpetuation of low level languages such as C into OOP is proof that the programming community is not willing to compromise, or sharpen its axe enough in order to bridge this costly gap.

The critique began with certain questions, and as no work can be absolute (particularly a programming language), it will end with more questions that it is hoped will create more debate, and more questioning into what we are really trying to achieve with program development.

Does C++ provide effective communication between programmers separated in both space and time? Does C++ provide communication between the levels of analysis, design, implementation and maintenance?

Are the compromises made by C and C++ still relevant to today's environments, and the environments of the not very near future?

Could C++ be regarded as the PL/1 of the object-oriented world, as PL/1 was the marriage of FORTRAN and structured ALGOL concepts, and C++ is the marriage of C with object-oriented concepts?

Are the compromises made for the restricted machines and environments of 20 years ago still appropriate for today? Are languages based on 20 year old compromises appropriate in modern software development environments?

Should new software developments be forced to accept such compromises?

Is C++ patching old material with new cloth, or pouring new wine into old wineskins?

What are we really trying to achieve in programming anyway?

Ian Joyner
November 1992

8. Bibliography

C++ ARM Ellis and Stroustrup "The annotated C++ Reference Manual" AT&T 1990.

[Capretz 87] Pierre J. Capretz "French in Action, A Beginning Course in Language and Culture" Yale University Press.

[Cline] Marshall Cline "C++ Frequently Asked Questions" comp.lang.c++ newsgroup.

[Crystal 87] David Crystal "The Cambridge Encyclopedia of Language" Cambridge University Press.

[DDH 72] Dahl, Dijkstra, Hoare "Structured Programming"

[Dijkstra 76] E.W. Dijkstra "A Discipline of Programming" Prentice Hall.

[Ellemtel 92] "Programming in C++: Rules and Recommendations" Ellemtel Telecommunication Systems Laboratories, Sweden.

[Ince 92] D.C.Ince "Arrays and Pointers Considered Harmful", ACM SigPlan Notices, January 1992.

[Mody 91] R.P.Mody "C in Education and Software Engineering" ACM SIGCSE Bulletin Vol.23 No. 3 September 1991.

[Reade 89] Chris Reade "Elements of Functional Programming" Addison-Wesley, 1989.

[RBPEL91] Rumbaugh, Blaha, Premerlani, Eddy, Lorensen "Object-Oriented modelling and Design". Prentice-Hall, 1991.

[S & W 79] William Strunk and E.B.White "The Elements of Style", MacMillan Publishing, 1979.

[Sakkinen 92] Markku Sakkinen "Inheritance and Other Main Principles of C++ and Other Object-oriented Languages". University of Jyväskylä, 1992. (Also published as selected papers in ECOOP '88, Computing Systems Vol. 5 No. 1, and Structured Programming Vol. 13 (1992).)

[SJE91] Saake, Jungclaus, Ehrich "Object-Oriented Specification and Stepwise Refinement" in IFIP Workshop on Open Distributed Processing Berlin, 1991.

[Weg91] Peter Wegner "Concepts and Paradigms of Object-Oriented Programming" ACM SIGPLAN OOPS Messenger Volume 1 no. 1 August 1990.

[X3J16 92] Members of the X3J16 working group on extensions "How to write a C++ Language Extension Proposal for ANSI-X3j16/ISO-WG21" ACM SIGPLAN Notices Vol. 27 No. 6 June 1992.

[Yoshida 92] Koichiro Yoshida Title and book in Japanese.