A C++ Critique

Introduction 1
C++ Specific Criticisms 3
- Virtual Functions 3
- The Nature of Inheritance 6
- Name Overloading 6
- Function Overloading 7
- A Note on Overloading and Nesting 9
- A Note on “A Discipline of Programming” 11
- Pure Virtual Functions 11
- Virtual Classes 12
- ‘::”’ ‘.’ and’->’ 12
- On information, computation and execution and paradigms 13
- Anonymous Parameters in Class Definitions 15
- Constructors 16
- Constructors and Templates 16
- Optional Parameters 16
- Bad Deletions 17
- Local Entity Declaration 17
- Friends 19
- Static 19
- Union 21
- Nested Classes 21
- Global Environments 22
- Header Files 22
- Class Header Declarations 23
- Calls on references to deallocated objects (Dangling Pointers) 24
- Type-safe Linkage 25
- C++ and the Software Lifecycle 25
- Reusability and Communication 26
- Reusability is a matter of Trust 26
- Concurrent Programming 27

The Role of Language 28
Generic C Criticisms 31
- Pointers 31
- Arrays 32
- Function Parameters 32
- void * 33
- void fn () 33
- fn () 33
- ++, -- 34
- Defines 35
- Case Distinction 36
- Assignment Operator 37
- Type Casting 37
- Semicolons 39

Conclusions 40

Introduction

This is a critique of the C++ language. It is an attempt to address the questions of whether C++ is a good OO language, can it be used to easily implement quick and small projects, how well does it scale up for large projects, and technical questions of how far it goes in supporting or hindering general good programming practices, and as a result the production of quality software? What is the relationship between a language and compiler, and software developers; and between the language and compiler, and the target system?

A programming language functions at many different levels and roles, and must be critiqued with respect to those levels and roles. Not only should a language be critiqued from a technical point of view, considering its syntactic and semantic features, but it should be critiqued from the viewpoint of its contribution to the entire software production process, how it facilitates communication between levels, and how the communication of ideas may be facilitated from one programmer to another, as often one programmer may not be responsible for a whole task over its lifetime.

The primary purpose of language is to facilitate communication, and a programming language must facilitate the consistent description of a system which satisfies the requirements of the problem at hand. It should be able to provide communication between programmers perhaps separated in space and time. A language should be able to express the intent of the systems designers and implementers. A language should also be able to provide mechanisms for project tracking, to ensure that modules (classes and functionality) are completed in a timely and economic fashion, and that the modules produced satisfy the original requirements.

A programming language aids reasoning about the design, implementation, extension, correction, and optimisation of a system. The preface to “The C++ Programming Language” has an interesting quote - “Language shapes the way we think, and determines what we can think about.” - B.L.Whorf. Does language shape the way we think? Does C++ shape the way we reason about programming? What is the role of language and what is the role of a computer language? Fundamental to the reusability of software is that subsequent programmers may be able to understand the nature of that software, by means of the communication medium and descriptive abilities of the programming language.

Programming is a dynamic activity. Program text usually does not stay static for long, especially during the development stages. Languages and compilers should provide consistency checks that aid in this dynamic process, ensuring that as program text is added or modified, previous work is not invalidated.

That a compiler may be able to translate the program into an executable entity is a secondary, and yet not unimportant consideration. A compiler has a twofold role. Firstly, to generate code that the target machine may execute, but secondly, and more importantly, to check that the programmers expression of the system is indeed a consistent expression. There is certainly not much point in generating code for a system, where parts of the description of that system has automatically detectable inconsistencies. The language definition provides the framework that makes this role of the compiler possible. The task of the machine is to execute the produced programs.

Programming languages should not be the exclusive domain of programmers, but should be tools in which design, and even requirements analysis may be achieved. Object-oriented techniques emphasise the importance of this, as abstract definition, and concrete implementation are separated, yet provided by the same syntax. Program text is the only reliable documentation of the system, so therefore, the language

should explicitly support the task of system documentation. As with all language, the effectiveness of communication is dependent upon the skill of the writer. Good program writers require languages which support the role of documentation. They also require that the syntax of languages is perspicacious, and easy to learn, so that those not necessarily trained in the skill of 'writing' programs, may be able to read them, and gain understanding of the system, just as those who enjoy reading novels are not necessarily novelists.

Can grafting OO concepts onto a conventional language realise the full benefits of OO? Perhaps a biblical quote can be applied: "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*

There is certainly a rush to the concepts of OO. OO has been a rallying point to which good software production practices have been gathered. These tended to have been lost in the structured programming world, as it lost its original focus (which was really object-oriented anyway). Now that OO has gained critical mass, it too is in danger of loosing its focus. OO extensions have been proposed for COBOL. One could speculate that there will be an OO BASIC.

How useful is the extension old technology languages to support some OO concepts, unless there is a wide scale abandonment of the concepts and practices in those languages that are contrary to the principles of OOP? How well can such hybrid languages be expected to support the sophisticated requirements of modern software production? Surely a basic premise of object-oriented programming is to enable the development of sophisticated software by the adoption of the simplest techniques possible, and that the software development techniques and methodologies do not impede the production of such systems?

All languages are affected by compromise and tradeoffs to some extent. To what extent do the decisions made affect the consistency of the concepts within the language, and the ease with which those concepts can be applied to any given problem? A language is not just syntactic constructs in isolation, but is the effects of the combination of all constructs. It is widely accepted that a good user interface will not give the user the option of choosing illegal actions, or combinations of actions. A poor user interface allows the user to do such things, perhaps correcting the situation with annoying error messages, or worse to execute an action that was not intended. To avoid error is better than to fix, (most people drive their cars with this principle in mind.) End users of computers understand such measures of quality, but programmers commonly accept analogous pit-falls in the languages and tools they use. Does the extra complexity disguise such problems, or preclude simple and consistent solutions?

A critique of C++ cannot be separated from criticism of the C base language, as it is essential for the C++ programmer to be familiar with C, and many of C's problems affect the way that object-orientation is implemented and used in C++. This critique may not be exhaustive of the weaknesses of C++, but it attempts to illustrate the consequences of these with respect to the timely and economic production of quality software.

The format of this report is that it is structured around technical considerations, with explanations and illustrations of how they affect the overall goals of software production and object-oriented programming. Firstly, it looks at criticisms that may be regarded as specific to C++, then at problems more general to C, and finally summarises with some conclusions.

The critique makes two general types of criticism. Firstly, there are safety concerns. These concerns affect the end user perception of the quality of the program. They can result in program crashes, or failure to meet requirements. Compilers can check for potential crashes, but they can also check for inconsistencies that may lead to failure to meet requirements. Languages provide the framework by which such compiler checking is made possible. Often this checking is enabled by requiring the specification of redundant information. Declarations themselves are an example of such redundancy. There is much redundancy in the source form of a program. The compiler uses this information to perform consistency checks, and then throws this information away, and produces as a result the executable system.

There is a misconception that safety checks are 'training wheels' for student programmers. This is quite an immature conception of programming. Languages such as Pascal were widely adopted by universities, because they were good languages. The best programmers realise that we are not good programmers, and as a whole, the computing community is still learning to program.

Secondly, there are 'courtesy' concerns. These concerns affect the view of quality of the program from within the development and maintenance processes. A fuller explanation of courtesy concerns is given later. Courtesy concerns are generally of a syntactic nature, whereas safety concerns are generally of a semantic nature.

C++ Specific Criticisms

Virtual Functions

Polymorphism and function name overloading are key concepts of OOP. A language designers choice is whether this should be specified in the parent or the inheriting class, in other words, is it the decision of the designer of the parent class, or the descendant class? There are cases, which can be made for both. Indeed both are not mutually exclusive, and can be catered for quite easily in an object-oriented language. There are three options:

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

2) a routine may or may not be redefined, descendant classes may use the routine as provided, or may 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, and not defined, no implementation is provided, and non-abstract descendent classes must provide a definition.

Cases 1 and 3 must naturally be decided by the parent class designer, and the language should provide appropriate syntax for these cases. Case 2 is the most common case and is the decision of the descendant class.

Option 1

In C++, the first case is not catered for. The closest is to not specify the routine as virtual. However, this means that the routine may be completely replaced, but is not virtual. This causes two problems. Firstly, it means that the routine may be unintentionally replaced in a descendent. This error should be averted by the compiler reporting a syntax error due to 'duplicate declaration'. This is logical, as descendant classes actually are part of the same name space as the classes that they inherit from, and the redeclaration of a name should cause a name clash within that scope. (See the section on name overloading for further explanations.) The second problem is illustrated by the following example:

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

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

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

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 presumably replaced, or added to the functionality of routines in class A for all objects in the subset class B. This example shows how C++'s virtual mechanism breaks a kind of safety. (This is like type safety as a routine not intended for objects of type B has been applied to an object of type B. But as objects of type B are also of type A, and A::nonvirt has been written for objects of type A, this cannot be strictly labeled type safety. This example shows that there is more to consistency and safety than can just be termed 'type' safety.) It could be argued that the C programmer knows what was intended, and this is how the language is defined, so there is no problem. If there was no other easy way to achieve the same effect then undoubtedly there could be no criticism. However, if the intention is that sometimes A::nonvirt and sometimes B::nonvirt be applied to objects of type B, the solution is simple, A::nonvirt and B::nonvirt must have different names. This solution means that all erroneous applications of the above construct will be caught. That way the compiler can be trusted to detect such potentially inconsistent situations. The problem therefore lies with the language definition. C++'s non-virtual overloading mechanism precludes this set of potential errors from being detected.

A further argument is that any statement should consistently have the same semantics. A typical object-oriented interpretation of a statement like a->f () is that the most suited implementation of f() is invoked for the object referred to by 'a'. In the above example, the programmer must know whether the function f() is defined virtual or non-virtual in order to interpret exactly what a->f () means, and 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 declaration or implementation DOES NOT change the semantics, of executable statements.

If a change in the declaration does change the semantics, then this should generate a compiler detected error, and the programmer must then be required to make the statement semantically consistent with the changed declaration. This reflects the dynamic nature of software development, where the program text is not static, but is subject to change, and therefore consistency checks are a necessary part of the role of a compiler.

For yet another case of inconsistent semantics of the statement a->f () vs constructors, section 10.9c, p 232 of the C++ ARM should be consulted.

Option 2

The second case should be left open in the parent class, and the decision left for the programmer of the descendant class, but C++ generally requires (although virtual can be introduced in an intermediate class in the inheritance chain) this decision to be made in the parent class, by the specification of virtual. This may be a problem in that it means that routines which aren't actually redefined in a descendant class are accessed via the slightly less efficient virtual table technique, rather than a straight procedure call. (Although this is never a large overhead, but object-oriented programs tend to use more and smaller routines, meaning that routine invocation becomes more of a significant overhead.) An optimising compiler and linker may replace routine invocations which do not in reality require the virtual mechanism by straight procedure calls. The policy should be that any routines which may potentially be redefined should be declared virtual.

A potentially more serious problem is that since the programmer of the descendant class does not specify that the routine is being intentionally redefined, the original routine may be redefined unwittingly. Object-orientation means that the same name can be used, but the programmer should be conscious that this is what they are doing, and state this explicitly. If the programmer is not aware of this then this should be a syntax error reporting that the name has already been defined. C++ adopted the original approach of Simula, and this has been improved upon, and other languages have adopted better, more explicit approaches, which avoid the error of mistaken redefinition.

Eiffel and Object Pascal cater for this situation by requiring the descendant class programmer to specify in that class, that redefinition is intended. This has the extra benefit that a later reader or maintainer of the class will be able to easily identify which routines have been redefined, and that this definition is related to a definition in an ancestor class without having to refer to ancestor class definitions. Another problem with the C++ scheme is the ability to reuse function names based on different function signatures. (See name overloading for a further and alternative explanation).

Option 3

The third case is catered for by the pure virtual function, which means the routine is undefined, and the class is abstract and cannot be directly instantiated. If a descendant class is to be instantiated, then it must define the routine. Any descendants that do not define the routine, are also abstract classes. There is nothing wrong with this in concept, but see the section on pure virtual functions for a criticism of the syntax.

Virtual is not an easy notion to grasp. The related concepts of polymorphism, redefinition, and overloading are easier to grasp than virtual, being oriented towards the problem domain. Virtual routines are the underlying mechanism by which redefinition in heirs is implemented. Virtual is an example of

where C++ obscures the concepts of OOP as the programmer has to come to terms with the low level concepts, rather than the higher level object-oriented concepts. Interesting as underlying mechanisms may 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 may prevent the adaptation of software to further advances in the underlying technology and execution mechanisms (see concurrency).

In summary: the concepts which should be expressed are, that a parent class should be able to prevent redefinition of selected routines, but in general, all other routines should be left open for redefinition, as prevention of redefinition is the exception, rather than the norm. Explicit syntax to provide the three options makes the programmers intent clear, and can avoid rare, but albeit subtle and possibly disastrous errors. To avoid mistaken name overloading, the redefinition of a routine should be explicitly stated in the descendant class. C++'s virtual mechanism may be used in such a way that a kind of safety similar to type safety, but not strictly type safety, is broken. A routine may be applied to an object of a class for which it was not intended. A compiler may determine exactly which routines need to be made or not made virtual.

The Nature of Inheritance

Inheritance is by nature a close relationship. Objects that are instances of a class are by definition also instances of all ancestors of that class. In order for there to be effective object-oriented design, the consistency of this relationship must be preserved. Consistency can be checked by ensuring that each redefinition in a subclass is consistent with the corresponding definition in the ancestor class. This means that the original design and requirements of the ancestor class continue to be met through successive levels of refinement. If the requirements cannot be met, then this indicates an error in the design or implementation, and indicates that rework is required. Thus the 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. Therefore C++ does not realise effective object-oriented design.

Name overloading

The choice of names is of fundamental importance in the production of self-documenting and understandable, and therefore reusable and maintainable software. Name overloading is a powerful, but problematic technique. The problem with it is that two entities with the same name cause confusion. This is easily illustrated by considering a group of people, where two or more have the same first name. This problem has been overcome in conventional languages by providing the notion of scope, where the same name can be used in different contexts. For example, brothers in the one family, are usually assigned unique names by their parents. When a name is used twice within the same context it should be a syntax error, due to duplicate definition.

In object-oriented programming, 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. What should happen to the scope of a declaration in a base class, in a derived class? There are three choices. Firstly, the name is in scope only in the immediate class where it is declared, but not the subclasses. Secondly, the name is in scope in a subclass, but the name may be reused, whether the declaration is virtual or not. Thirdly, unless the name is reused in the context of

reimplementation of a virtual routine, the reuse of the name is reported as a syntax error. This relationship of name scope is obviously not symmetric, as names in a subclass will not be in scope in a superclass (or references to objects of the superclass, although this is not the case in typeless languages such as Smalltalk.)

The first case precludes software reusability, as subclasses will not inherit definitions of implementation, so case 1 is not worth considering. The second case is C++'s approach. There are two potential problems which may arise. Firstly, the name may unintentionally be reused, as the compiler does nothing to stop this. Secondly, the parameters being passed to the routine cannot be type checked against the original routine, as there is not assumed to be any relationship between the original, and the new routine. (See the note on name overloading and nesting for a further explanation of the problems of scoping vs inheritance). Since consistency checks between the superclass and subclass are not possible, the tight relationship that is implied by inheritance, which is fundamental to the design of object-oriented systems, is not guaranteed. This can possibly 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.)

In order to provide the consistent customisation of reusable software components, object-oriented programming permits the same name to be used, only by the redefinition of the original entity. The programmer of the descendant class should tell the compiler 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.)

In object-oriented programming, two entities with the same name may be used in the same context. This conflict must be resolved by qualifying the names, so that the correct entity is referenced. Names must be qualified with the object to which they belong. Thus two entities named 'field' may be distinguished by 'a.field', and 'b.field'. This is a bit like in a group of people with the same name, qualifying which person is intended by adding their family name.

In summary: The relationship of inheritance implies a tight relationship between base and derived classes. If consistency checks are not satisfied, then this is a good indication that the design of the system is incorrect, and detection of such faulty designs at an early stage will save much wasted work. C++ fails to provide an important class of consistency checks.

Function Overloading

C++ allows functions to be overloaded if the arguments in the signature are of different types. This has examples where such overloading can be quite useful, for example:

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

will ensure that the best routine for the types int and real will be invoked. However, object-oriented programming actually 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) will give syntax errors, as the types of the arguments do not agree. (By operator overloading of course, these may be better expressed, i max j and r max s, but min and max are peculiar functions that may want to accept two or more parameters of the same type.)

Now the above shows that in most cases, function overloading can be expressed consistently within the object-oriented paradigm, without the need for the function overloading of C++. However, C++ makes the notion more general, as the function may be overloaded by more than one parameter, not just the implicit current object parameter. The generalisation of any technique is often a good thing, as it removes restrictions. In this case it may provide a useful mechanism, but how many examples require this flexibility, and how could they be programmed in other ways?

However, in order to provide such flexibility introduces several potential errors which the compiler will not be able to check, as it must assume that what may be inconsistencies, are correct, in order to allow such overloading to be provided.

If the programmer intends to redefine a virtual routine, but makes a mistake in the declaration of the function signature, then an entirely different function will be assumed to be defined by the compiler, and any calls to the function using one or other of the signatures will also fail to detect the inconsistency.

If the programmer does make a mistake in supplying the parameters on a call to the routine, a C++ compiler will not be able to be specific about the error, all it can report is that no function with a matching signature could be found. Presumably, if the programmer has made this sort of mistake it is for subtle reasons, and may be difficult to actually pinpoint which parameter was at fault. Secondly, the incorrect parameter may accidentally match, one of the other routines. In which case this error will be propagated into the production code, and may not be tracked down for a long time.

If it is felt though that C++'s scheme of having parameters of different types is useful though, it should be considered that object-oriented programming does provide this but in a more restricted and disciplined form. This is done by specifying the parameter as needing to conform to a base class, and then any parameter passed to the routine may be a type of the base class itself, or any class derived from the base class. For example:

```
A.f (B someB) {...};

class B ...;
class D : public B ...

A a;
D d;

a.f (d);
```

This means that strict type checking of parameters is possible, as the entity 'd' must conform to the class 'B'.

The alternative to function overloading by signature, is to require all names to be unique. Names should be the basis of distinction of entities, and is the purpose of naming. This is known to work, solves the

above problems, as the compiler can cross check that the parameters supplied are correct for the given routine name, and furthermore will result in better self-documented software. Often it is difficult to choose appropriate names for entities, but this is well worth the effort. Philosophically, a name is most important in describing the characteristics of a person or entity. Names give things their identity.

In summary: The relationship of inheritance is a strong relationship. It is the basis of reusability, and the ability to be able to assemble software systems out of preexisting components. Such assembly requires that consistency between the components being assembled must be checked. Assembling a jig-saw requires that not only the pieces fit in their cut out shape, but that the resultant picture makes sense. Assembling software components is more difficult, as unlike the jig-saw, which is reassembling that which was complete before, assembling software components is building a system that has never existed before. C++ has not implemented the notion of inheritance in a manner that fully exploits the advantages of inheritance, with regards to software reusability. This may lead to the opinion that C++ is not a truly object-oriented language, as it does not ensure that programs written in the paradigm are consistent with the paradigm. Certainly, there are not very many reusable C++ libraries available, which tends to suggest that C++ may not support reusability as well as possible. This means that one of the fundamental goals of object-oriented programming has not been met.

A note on name overloading and nesting

Name overloading should be provided and used in ways that are explicit and expected. Object-oriented programming provides name overloading by inheritance and redefinition. Routines provide name overloading, by local declarations of names being limited in scope to the routine of declaration. Thus the same name may be used in other routines, without any clash in the name space. This provides an important benefit, the prevention of name clashes, and enables two programmers to independently write routines without being concerned that the other programmer is using the same name, so that the modules may later be combined without the programmers having to worry about name clashes. Nesting of routines, or blocks has always had a problem with overloading, in that if a name is redefined in an inner block, then it may mistakenly override the definition of the name inherited from an outer block, and the inner block may not access the entity of the outer block, unless some extraneous entity qualification mechanism is introduced into the language (which most languages chose not to provide.)

This report suggests that restriction to only these two mechanisms (redefinition in subclasses, and local scope to routines) is necessary to gain the maximum benefit from name overloading. This is reasoned by common sense, but the proof of this is in the realm of the language theorist, and beyond the scope of this document. The exclusion of all other forms of name overloading results in no loss of generality, and only the slight inconvenience that the programmer has to choose unique names, as the compiler will report name clashes as a syntax error due to ‘duplicate identifier’. Note this is only a potential run time error, if this level of safety is not provided. An automated tool such as a compiler has no way of guessing the programmers intent, and therefore is bound to report this as an error, as it could be an inconsistency, not because it is an inconsistency. The use of name overloading in such forms is frequently found to be the cause of subtle errors.

It is also suggested that contrary to the usage in most high level languages, that a name should not be overloaded, while it is in scope, and that the reuse of a name in nested procedures and blocks should result in a syntax error. In fact textually nested blocks ‘inherit’ the named entities of outer blocks, and perhaps as argued for virtual, explicit redefinition of names should perhaps be provided to avoid error. However,

this actually provides no or very little benefit, unlike redefinition of inherited entities in the class sense. Nesting automatically makes entities tightly coupled.

Consider another possible error that can happen due to nesting. If an overloaded nested declaration is removed, then references to that name will not result in syntax errors due to their being the same name in the outer environment. An instruction mistakenly not removed that changes the value of the entity, will now mistakenly change the outer entity. Consider:

```
{
     int i;
     {
            int i;
            i = 13;
     }
}
```

Now delete the inner declaration:

```
{
     int i;

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

Such nested blocks as above are also contrary to the principles of object-oriented programming, as inner blocks are ‘tightly’ coupled to outer blocks, and inner blocks have access to the entire environment of the outer block, instead of through well defined interface routines. Nested blocks are the opposite of the loosely coupled software components, which are the ideal of object-oriented programming. Nested blocks are similar to parent and child classes as a child inherits entities defined in a parent, and an inner block inherits entities defined in an outer block. This is why textual nesting is not required in object-oriented languages, as the ‘nesting’ mechanism of inheritance is more sophisticated.

The above example also shows that a program text is not a static entity, but a dynamic one. Programmers may easily make modifications to program text that violates the consistency of the original system. This scenario happens often, as a programmer may come back to program segments written some time before, so that some or many of the original details are forgotten, or modifications may have to be made by a completely different programmer. These scenarios are very common today, especially with large projects, and with software that evolves over time to meet new and changing customer requirements. In this case it is vital that the language and compiler system be trusted to detect any possible inconsistencies that may be introduced into the system due to modifications.

The restriction of concepts to their useful forms, is only a restriction on the language, and coincidentally have the effect of keeping the language simple. Such restrictions do not in general restrict the programmer in the production of flexible, and efficient software, but aid in the production of such software, by avoiding subtle errors which may otherwise occur. Such restrictions in the language and compiler in fact aid flexibility in the software process, as programmers can more confidently make changes, and rely on the compiler to make sure that the original programmers intent has not been contravened.

The most useful tools in any trade are those that do exactly what is needed and nothing more. Name overloading is just such a tool. It is very useful, but where it is provided in more than its most useful forms, the less useful forms 'blunt' the effectiveness, and even work against the usefulness of overloading, as confusion of names becomes greater, meaning that name clashes and inconsistencies of use cannot be detected. Taken to an extreme, name overloading may mean that one name can be used for every entity. Then we may as well do away with naming entities altogether. After all, names are not needed in order to program, the compiler just throws them away in the resultant code.

In summary: Name overloading is good. Name overloading is bad.

A Note on "A Discipline of Programming"

Having written the previous section, 'An essay on the Notion: "The Scope of Variables"', in E.W. Djikstra's "A Discipline of Programming", was revisited, to find very similar reasoning about declarations in nested situations. The following points come from his argument.

Programming techniques facilitate the management of extremely large state spaces. There are two notions here, of how to manage the state space, and how to manage change within the state space. This is precisely the activity of programming.

State spaces may be managed by division, and nomenclature (naming entities), within those divisions. The activity of division must cleanly divide the state space into independent entities, and nomenclature must provide a consistent system, where name clashes, interference, and confusion are not possible. A scheme that implements this will necessarily be as simple as possible.

State change within a state space, must also be provided in the simplest possible manner, and one such mechanism is available, and has been used successfully in many programming languages, the assignment, which changes the values of variable entities within a state space.

These notions are the basis of keeping programming a simple and manageable activity. The larger state spaces (systems and programs) grow, the more necessary it is to observe the simplicity of software organisation. It is the purpose of programming languages to implement these concepts in the simplest, and therefore the most manageable forms.

Pure Virtual Functions

As mentioned above, pure virtual functions provide a means of leaving a function undefined and abstract, which also means, that a class with such an abstract function may not be directly instantiated, as the function must be defined in a descendant class. The concept is quite alright, but the syntax to express this concept illustrates where C++ misuses mathematical notation to express a natural concept. The syntax is -

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

This syntax on the face of it leaves the reader to guess as to what its meaning is, even those well versed in object-oriented concepts. The choice of designing this construct is to use some indirect mathematical construct as C++ has done, or to use a keyword, which directly expresses the concept. Direct expression of concepts enhances communication, and the ease with which a language may be learnt. The C++ decision

is in keeping with the C philosophy of avoiding keywords, but often this avoidance is at the expense of clarity. This concept would have been more clearly implemented by a keyword, such as -

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

or

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

A suggestion that the mathematical notation makes is that values other than zero may be used. What if 13 is equated to the function? -

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

A simple suggestion to fix this is to define '= 0' as abstract:

```
#define abstract = 0
```

then

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

Virtual Classes

If class D multiply inherits class A from classes B and C, then if it only wants to inherit a single copy of class A, the inheritance of A must be specified as virtual in both B and C. Two questions arise from this. Firstly what happens if A is only declared virtual in only one of B or C, and secondly, supposing a class E wants to inherit A via B and C by a different policy, ie multiple copies of A, than in D. In C++, the virtual decision must be made early, reducing the flexibility which may be required in the assembly of derived classes. In a shared software environment, as classes B and C may be provided by different suppliers, it should be left to the implementer of class D or E, exactly how to resolve this problem. Such flexibility is key to reusable software. In principle you cannot envisage when designing a base class all the possible uses in derived classes, and the attempt to do so considerably complicates design. C is supposed to trust programmers, and in unimportant ways gives them flexibility, but the design of C++ restricts flexibility where it actually is important, especially in the object-oriented sense.

'::', '.' and '->'

While the '.' and '->' syntax came from C structures, it is heavily used in C++ in accessing class members, and illustrates where the C base adversely affects the flexibility required of an object-oriented language. When the programmer declares an entity, a decision is made about how the entity will be accessed. The declaration decides whether the entity is accessed directly, or via a pointer. When members of the entity are accessed, the programmer should not be required to know how exactly to access them, but should access them in an implementation independent fashion, without the need to know whether the entity is being accessed directly, or via a pointer. This is because the compiler can determine exactly how to access the entity.

In more detail, C++ x.y means access the member y directly in the structure or object x, whereas x->y (or the equivalent *x.y) means access the member y via the pointer x. One of the key concepts of object-

oriented programming is that access to entities is independent of implementation. The realisation of implementation independence is a simple matter for compiler technology, as the compiler knows from the declaration of ‘x’, whether x is an object itself, or a pointer to an object. The compiler may therefore be relied upon to generate the correct access code. Thus only one access operator is required in a high level language, not both ‘->’ or ‘.’. This illustrates why declarations are important for implementation independence, and why type safety is also important.

A further problem is that the distinction reduces flexibility in the software fabrication process. If it is required to change an objects status from via a pointer to direct, all ‘->’ dereferences must be changed to simple ‘.’. This discourages the optimisation of changing an entity to direct access, although the correction of changing a direct access to access via a pointer may be forced.

For declarations, C++ requires the use of the ‘::’ form. Most languages use one syntax (typically the dot) for both declarations and access. ‘::’ is also defined as the scope resolution operator. (Interestingly, in an early paper, ‘Classes: An Abstract Data Type Facility for the C Language’, 1981 by Bjarne Stroustrup, ‘.’ was used instead of ‘::’.) Scope resolution is a difficult problem with regards to the inheritance of features from a common ancestor due to multiple inheritance. It would be more advantageous if such accesses were provided by one consistent syntax. (Scope resolution by the ‘::’ operator is rather like qualification by ‘qua’ in Simula.) It is better to make decisions of this nature in declarations, and rely on the compiler to generate the different code for different accesses, as if the declaration policy ever changes, then all that is required is one recompilation, rather than any rework of the program text by the programmer.

In summary: access to members should be provided by the same syntax, as this ensures implementation independence, and thereby enhances programming flexibility. A change in access policy may then be effected by the change in the declaration only, and simple recompilation. This is enabled by compiler technology, which means the compiler can generate the correct code for the given semantics, therefore relieving this burden from the programmer.

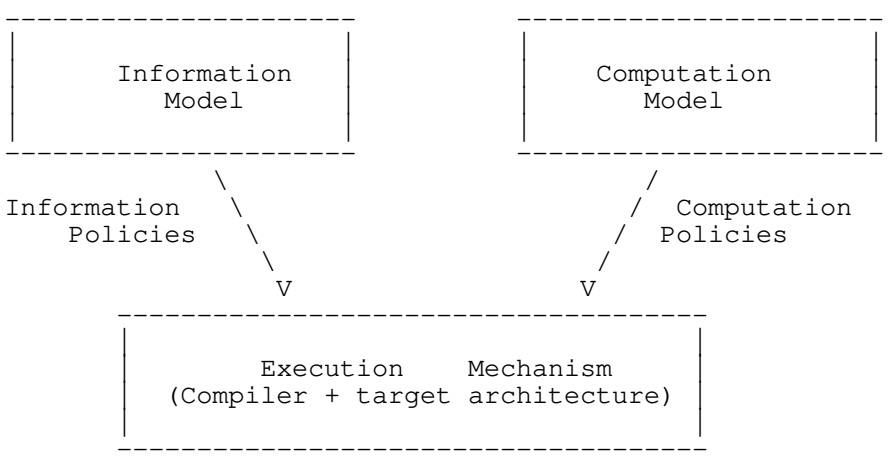
On information, computation and execution and paradigms

The role of programming may be seen as the separation of concerns. This separates information modeling from the specifics of computation, and thereby, computation from the specifics of execution. This model of programming ensures data independence, and therefore flexibility and portability in software production, and the above argument, and others in this paper are a consequence of this. The information model, can be seen as setting up policies of data form, and type, and the interfaces and operations on this data. The computation model can be seen as specifics of the computations that can take place on the information.

The execution mechanism is a combination of compilers and the target architecture and environment. The execution mechanism is responsible for the combination of policies from the computation domain, and the information domain, into an executable system. For example, if the computation ‘a + b’ is to be executed by the execution mechanism, the execution mechanism must determine whether a and b are integers or reals, and actually invoke the correct mechanisms to execute the add based on those types. Similarly, if ‘a.b’ is specified in the computation domain, the execution mechanism must look up the access policy on a, in the information domain. If a is an object, then ‘a.b’ is executed in C terms. If a is a pointer to an object, then ‘a->b’ is executed in C terms. In a compiled system, these decisions are mode early, at

compile time, so there is no run-time overhead. In an interpreted system, these decisions are made at run-time resulting in some overhead, but with the advantage of extra flexibility.

A schematic of the information, computation and execution mechanism looks like:



The information and computation policies are determined by requirements analysis, design, and implementation. This separation of concerns, makes the information, and computation aspects of a program independent, and this ensures flexibility. Interestingly, ISO's Reference Model for Open Distributed Processing describes a similar model in order to ensure distribution transparency of applications and services within a network.

Another way to look at this is that the information model provides a schema, by which the computational model is interpreted to be meaningful, and to which the computational model must conform. It is the compilers job to ensure that the computational model conforms to the information model. In natural language, we use schemas frequently. For example, the sentence "The boy drank the computer and switched on the glass of water" is grammatically correct, but the computational model of verbs does not conform to the information picture that we understand about computers and glasses of water.

Now programming has much more precisely defined semantics than natural language, so naturally we should expect to find many more such schema checks within programming languages.

There is another concept in object-oriented programming which is closely related, that of paradigm. Object-orientation is a paradigm. If the object-oriented paradigm is strictly implemented, there are also many consistency checks that a compiler may apply, similar to those of schema checking. Between the dual concepts of schema and paradigm, a solid and disciplined framework for program development is provided.

Multi-paradigmed approaches have also been proposed, to bring the advantages of several paradigms into one framework. But multi-paradigmed approaches have their own problems, and haphazard mixing of paradigms can lead to inconsistencies between the paradigms. Indeed if two or more paradigms are badly mixed, we may end up with something of little advantage. At best multi-paradigmed technology is immature. To get any advantage of a multi-paradigmed approach, paradigms will have to be carefully

combined, and not necessarily equally. It has been suggested that C++ can be used for many different styles or 'idioms' of programming. This is very similar in concept to multi-paradigmed programming, but it has been recognised that bending languages to resemble other languages by defines and other techniques is problematic, and the Swiss Army knife approach cannot be as efficacious as a tool which is specifically designed for the purpose of application. By this approach, syntactic constructs similar to those used in the paradigm may be built, but in general checks to make sure that the constructs are used consistently with the paradigm are impossible.

One of the strengths of the object-oriented paradigm is that it is excellent for the large scale organisation of software. Therefore, the object-oriented approach may be most useful for providing the overall structure for systems written in other paradigms. The object-oriented paradigm may provide the framework, within which other paradigms, such as logic, procedure, functional, and machine oriented paradigms can exist. Thus an object-oriented framework will provide an object-oriented wrapping for embedding other paradigms. One example is Apple's MacApp, which provided an object-oriented wrapping around the Macintosh Toolbox paradigm. Machine and operating system paradigms may also be wrapped in object-oriented interfaces, thereby centralising all environment dependencies in one or as few places as possible.

Anonymous parameters in Class Definitions

In C++ parameters in function templates are not required to be named, but just the type alone may be specified. For example a function f in a class header can be declared as f (int, int, char). This gives no clue to a class user as to the purpose of the parameter, without referring to the body of the class implementation. Meaningful identifiers are essential in this situation, as this is the abstract definition of a routine, and a client of the class and routine will need to know that the first int represents a 'count of apples', etc. The use of anonymous parameters handicaps the purpose of abstract descriptions of classes and members: to facilitate the reusability of software. Program text is the medium by which the meaning of the system is captured for some future activity, short or long term. Communication of intent of a software element is essential if reusability is to be achieved. A compiler strips away this level of communication, producing a machine executable entity. The careful production of a semantic entity should not be penalised by languages or compilers that perform less than optimal translations. But neither should a language definition allow less than optimal expression to the human reader.

The reason that anonymous parameters may be in C is that it is not necessary to specify a name in a function template. But then naming as has been pointed out is not at all necessary in programming. Naming only exists to help the human reader identify different entities within the program, and to reason about their function. To this purpose naming is essential. Anonymous parameters may save typing in a function template, but then programming is not a matter of convenience, especially, where such a convenience is not a matter of convenience for later readers. This is an example of where redundancy is beneficial, as it saves a later programmer from having to go and 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).

The issue of anonymous parameters may be argued to be a safety issue, as a programmer may guess at what the purpose of a parameter is from the type, but guess wrong. However, it is often counter argued that C programmers always know what they are doing, and don't make this sort of mistake. If we ignore then that this could be a safety issue, it can still be labeled a 'courtesy' issue. Courtesy issues become even

more important in the context of reusable software, as an interface client must know what the intention of the interface is for it to be used effectively. A courtesy implies an inconvenience to the one who provides the courtesy, but provides a convenience to one or more others. Courtesy issues include choosing meaningful identifiers, consistent layout and typography, meaningful and non-redundant commentary, etc. Courtesy issues are not just a style consideration, but should be reflected in, and supported by language design. A language may support, but 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 not being careful about the languages that we develop and choose for software development.

In summary: the purpose of programming is to capture the semantics of a system. As a matter of style, meaningful identifiers should always be used in function templates in C++. Reusable software requires that the semantics of the abstract system are clear to enable that system to become a subcomponent of some other system. Courtesy issues are important if reusable software is to be realised, and should be considered in the design of a language.

Constructors

Multiple constructors are allowed by having different arguments, as for functions. This has the disadvantage that it precludes having two different constructors with the same arguments. Such constructors may be useful for initialising the object in different valid states. 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 behind the different constructors. 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.

Declaring named entities is a redundant activity, but it was realised in the transition from FORTRAN to ALGOL that such redundancy provides the mechanism which enables compilers to detect certain classes of errors. Naming is essential in the management of state space of a program, so the C++ implementation of constructors is at best an example of dubious merit.

Constructors and Temporaries

When can ‘return <expression>’ result in some value other than the result of the expression being returned to the caller. 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.”

Now putting this together, creation of a temporary is implementation dependent, so may or may not be done. A constructor is called as a side effect, and this can completely change the value held by the object if it wishes. But as different implementations may decide to do different things with temporaries, this means that the potential exists that different results may result from different C++ implementations.

Optional Parameters

Optional parameters which 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 may be convenient in shell scripts, and interactive systems, but in large scale software production, where precision is mandatory, can lead to ambiguities and mistakes. Firstly, optional parameters open up the possibility that the programmer may assume the wrong default for a parameter. Secondly, it is possible that the programmer did not intend to omit the parameters (most errors being, what the programmer did not intend), but optional parameters mean that the compiler must assume that this is correct, and this bypasses the safety check that parameters must match exactly those defined in the function header.

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.

This mechanism can be easily provided by other means, already in the language. For example, you can make a parameter optional by having a call to another (inline) routine which provides the default. This not only provides the convenience of optional parameters, but is more powerful, as any parameter or combination could be filled in with any combination of defaults, not just the last parameters. Multiple sets of defaults may be provided by multiple intermediate routines.

In summary: optional parameters provide little convenience, but open up the possibility for errors. Every convenience, and more, of optional parameters may be provided by other language mechanisms. In short, use these alternative mechanisms in preference to optional parameters.

Bad deletions

The following example is given on p.63 in the C++ ARM as a warning about bad deletions which 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, which are ill-defined language constructs, which can only be covered by

warnings of potential disaster. To avoid such language deficiencies would result in loss of compatibility with C, but this may be a good thing if problems such as the above disappear. But then the resultant language may be as far removed from C as to suggest that C is abandoned altogether.

Local entity declaration

The ability to declare an entity close to where it is used, has both advantages and disadvantages. It is a convenience, as you can declare the entity close to where it is used, but this can make a routine appear more complex and cluttered. Another potential problem is that an identifier can be mistakenly overloaded within a nested block in a function, with the resultant problems covered in name overloading and nesting. This simple form of name overloading was used in ALGOL (name overloading has been around a long time, in fact the designers of Simula realised this, and hence object-orientation was born, although they thought it was just structured programming if you read the book by Djikstra, Hoare and Dahl). As has already been pointed out in the section on name overloading, overloading is best provided in limited by well controlled forms. C originally avoided this problem by not having nested routines or blocks, (a block in the ALGOL sense contains both declarations and instructions.)

It should also be of concern that the local entity declaration scheme of C++ does not ensure the ‘separation of concerns’, as described in the section of information, computation, and execution mechanisms, giving the benefits of data independence, and implementation transparency. This is important in large scale software development, and protection of programming investment, should the requirements of the system be changed.

The ARM should be consulted for further problems of local declarations vs branching, which shows the complications in intermingling declarations and instructions. Again fixing poor language definition by caveats cannot make up for the faulty definition.

The C++ FAQ (Q83) is not very clear 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 subject of a ‘new’ instruction. In well written object-oriented software, routines will be small, typically performing one atomic action per routine*, and therefore unneeded entities will not be constructed until the routine is actually called. This strategy means the provision of such locals is less likely to be of any benefit. If as the FAQ suggests, that initialising an object is so expensive, then the overhead of a routine call will be negligible, thereby realising the efficiency gain by good modularisation by subroutines.

Further if an object is in the stack, this suggests that its constructor should be simple, only initialising the object to some neutral state (like integers to 0, booleans to false, etc), leaving the bulk of the work to where the object is actually used. Thus a complex constructor can be broken up into a simple part, which just gets the object going, and a more complex part in a normal routine, which is called closer to the time of use. On the whole, this seems to be a better modularised approach. In fact allocating all locals at the beginning of a routine may be an optimisation on some processors (like the M68000), where the stack space for all locals is allocated in one instruction by adjusting the stack top, rather than many little increments for each individual declaration statement.

Other than that, the FAQ offers some excellent general advice on optimisation in this section, as you should not embark on expensive creation operations before doing some cheap checks to see whether that is necessary. Also the criticism in the FAQ of 'set' methods is quite useful.

*Small routines which implement atomic operations are fundamental to loose coupling. For example, a base class that provides a routine that logically performs operations A and B, is not useful to a subclass which wants to perform A as it was in the base class, but needs to provide its own implementation of the logic of B. This way the descendant is forced to reimplement the logic of both A and B, missing an opportunity to reuse the logic labeled A. Such tight coupling is to be avoided, as it reduces flexibility.

In summary: due to the typically small routines written in object-oriented systems, declarations intermingled with instructions result in very little convenience, and if the system is designed correctly, will probably not be used a great deal. Local declarations lead to complications, and also can result in unintended name overloading, due to nesting. Use of local declarations means that information and computation concerns are not cleanly separated.

Friends

Friends provide a mechanism for overriding data hiding. A friend class or function of a class has access to the private data of a class. Friend is a 'limited export' mechanism. The friend function or class has access to all the members in the private section of a class. There are three problems with friends:

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

The usefulness of friends shows that there is a case for shades of grey between public, protected and private members. The functionality of friends may be provided in ways that do not have the above three problems, by ways which are consistent with the object-oriented paradigm, while not requiring the introduction of a foreign concept such as friend. The programmer may make careful use of such mechanisms by careful class design, which leads to as little coupling between components as possible, in conjunction with a selective export mechanism. A selective export mechanism (more general than public, private, protected and friend), will also document the couplings between entities that can occur in the system, as does friend. Selective export specifies that not only should a member be exported but to which other classes specifically to which it is to be exported.

One reason given for the use of friends, is that it allows efficient access to data members, which would otherwise have to be accessed via a function call. The way C++ is often used is that data members should not be put in the public section, as this breaks the data hiding principle.

Data hiding may be better described as 'implementation hiding'. The interface to the external world, exports entities that return values of given types, so there should be no restriction on the export of variable members. The strategy is that entities should not be exported if they expose any detail of implementation, and that exported variables are treated by the outside world in the same manner as functions, ie they cannot be assigned to. This is because, when used in expressions, there is no semantic difference between

functions and variables, as 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.

In summary" The concept of friend may be provided more generally and consistently by selective export, which specified which entities are to be exported, and to which classes.

Static

Static is a word that causes much confusion in C++. This confusion is mentioned on p. 98 of the C++ Annotated Reference Manual (ARM), which claims it has two meanings. Actually it has three meanings. Firstly, as above, a class may have static members, and a function may have static entities. The second meaning comes from C, where an entity declared static is local in scope to the current file. The choice of independent keywords would easily solve this trivial problem. The third meaning is a general object-oriented meaning, not specific to C++. It is that objects may be statically allocated, as opposed to dynamically allocated in free space, or automatically allocated and deallocated on the stack, when a block is entered.

Static class members are a useful concept, but of limited use, and the functionality can easily be provided without the extra notion of static. On page 181 of the ARM, the reason is given that statics will reduce the need for global variables. Such members though may be placed in a class of their own, and one copy only of this class is instantiated (a global object). Each object is instantiated with a reference to the object with the static data. For example

```
class static_data
{
    // Only one object of this class is instantiated.

    ... static member declarations
}

class original_class
{
    static_data * our_statics;
}
```

This is a clue as to why global entities are unnecessary in object-oriented programming. (See the note on name overloading and nesting, and global environments.)

Static may also be applied to declarations in functions. These are not needed in an object-oriented language. The reason and history is this. ALGOL had the notion of 'OWN' locals in blocks. The semantics of an OWN entity was that on every entry to the block, it preserved the value from the last entry to the block. The implementation of this was that at run time, it was kept in the global environment, but at

compile time, the scope of the OWN entity was limited to within the block in which it was declared. Now this caused complication in recursion, as the same instance of the variable was used in all invocations of the procedure, rather than each invocation using separate local storage on the stack.

The designers of Simula generalised the ALGOL notion of block, so that instead of deallocating the storage on block exit, that storage could be made 'persistent', and so object-oriented classes were born. Now since the declarations in this 'class' block were no longer deallocated, this removed the need for the OWN concept, which had proven problematic in ALGOL, so there is no OWN in Simula. So there is no reason for static or OWN declarations in object-oriented languages, as this is achieved by making the static or OWN a class member, if persistency is required in the lifetime of the object, or if this value is required to be shared among objects, then it may be provided by a shared object, as suggested for static class members. If persistency is required for the lifetime of the program, then this shared object will not be deleted until the program is terminated.

In summary: the static keyword and concept has been confused by overloading one word with different semantics. The object-oriented concept of class is very simple, yet very powerful. It is the basis for a consistent division of a programs state space. The power of this method is that other mechanisms for storing permanent data in a running program are not needed, and only complicate systems. The mechanisms that are not needed are globals, static members, and static declarations (OWNs).

Union

As with static, this is another construct not needed in OOP. Firstly, similar concepts have been recognised as problematic in other languages, for example, FORTRAN's equivalences, COBOLs REDEFINES, and Pascal's variant records. Such mechanisms when used for overloading the same memory space forced the programmer to think about how achieve this. The stack mechanism in recursive languages meant that this was not necessary, as memory was automatically overloaded for the locals of procedures, and the same space was reused as procedures were entered and exited. So this use of union when used similarly to FORTRAN's equivalences is not needed.

Union is also not needed to provide the equivalent to COBOL REDEFINES or Pascal's variants. In OOP though, this is provided for by inheritance. A reference declared as a superclass may be used to refer to all subclasses, and this way provides the same semantics as union, only in a type safe manner. A declaration of a reference to a class is implicitly a union of all subtypes of the class.

Nested Classes

Nested classes were provided in Simula, in much the same way that ALGOL provided nested procedures. This was later realised to be of questionable value, as it was not in the free spirit of object-oriented decomposition, where classes should be loosely coupled, thereby supporting software reusability. C indeed did not have the complexity of nested functions, but C++ has chosen to implement this complexity for classes, which is of less use than nested functions. Nesting is achieved more generally by inheritance. In C++, not only may classes be nested within other classes, but they may also be defined within functions, thereby tightly coupling a class to a function. The concept of object-oriented programming is that the class is the fundamental structure, and that nothing has existence separate from class (including globals), but C++ is confused as to whether it is procedure-oriented or object-oriented. While the ability to able to have

classes local to functions may not cause any practical problems aside from precluding the class definition from being reusable, it causes theoretical and conceptual problems, which should not be ignored as nested classes are of little practical use. Of course examples may be given, but such examples may also be structured in simpler, more straightforward ways, without loss of generality. (See also the general comments on name overloading and nesting.)

Classes that can be nested within other classes and functions also raise questions about the usefulness of this approach to multi-paradigmed programming. It seems that the object-oriented approach will be most useful if it is used for the overall organisation of software, encapsulating other paradigms if necessary. However, C++ has combined paradigms in such a way, that object-oriented encapsulation, and the advantages that it realises are no longer guaranteed. Of course it may be pointed out that there are examples that may be written using the C++ approach, but maybe such examples may be better left to academia and research, rather than trade off consistency, which will be most beneficial for software production houses, which require commercial grade tools, rather than a general purpose research harness.

Global Environments

C++ functions may 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 the production of reusable software, as it is difficult to remove the class from its surrounding environment. This is a drawback of both global and nested environments. In fact the issue of side-effects is the most consistently addressed issue in the OO world.

A good OO language will permit changes to an objects state to be made only by routines in that object. Removal of a global environment is a technically trivial exercise. It is simply the requirement that the global environment be encapsulated in an object or set of objects of its own, after all that is what objects do: provide an encapsulated environment, and there is no reason why the global or external environment should not be encapsulated. Such objects facilitate a clean object-oriented interface to the environment external to the 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 therefore easily adaptable to new environments and projects. Software written in C++ has the Unix style environment interweaved throughout programs. This is not the way to open systems and cross platform development.

(See also the note on name overloading and nesting.)

In summary: The object-oriented concept of class is very powerful, and changes the way software should be structured quite radically. Globals are problematic in that they provide a mechanism whereby everything has free access to them, rather than the disciplined access of class. Since globals are just an environment, it is simple to encapsulate this environment into an object or objects of its own.

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 may be both derived from one source. In C++ though, programmers must maintain the two sets of information. The drawbacks of

replicated information are well understood. In the event of change, both copies must be updated, which could lead to inconsistencies, which must be detected and manually corrected. It is possible to provide tools that will automatically extract the abstract class header definitions from class implementations, and guarantee consistency.

Also, the programmer must consciously make those headers of imported classes available. If the correct #includes are not specified, this results in a round of syntax errors. Another problem is that if header A includes header B, and header B includes header A then a circular dependency is set up. The same problem also occurs if header A includes headers B and C, and header B also includes header C. This is overcome by the simple but messy fix in all headers of

```
#ifndef thismod
#define thismod

... rest of header
#endif
```

Headers show how C++ addresses the problem of independent modules by a non-object-oriented approach which is sub-optimal as it requires the programmer to supply this information manually. A class interface is the OOP equivalent of a module header. 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 program text, and functionality of tools like ‘make’ can be integrated into the compiler itself, and the errors encountered in the use of ‘make’ are completely avoided.

The mechanism for getting the header file information into a module is #include. #include is an old and not very sophisticated mechanism for the implementation of modularity. 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, and in Eiffel the unit of modularisation is the class itself, and all includes are handled entirely automatically. An OOP class is the more modern and sophisticated way of modularising programs, so there should be no need to support the #include mechanism in C++, at least for class dependency analysis, although required in order to remain compatible with C. Multiple inheritance is a form of #include.

To clarify the relationship between modules and classes further, a module contains data and routines that manipulate that data. A system is assembled by combining modules. A class also contains data, and routines that act on that data. An object-oriented system is assembled by the combination of classes. Modules are a primitive form of classes. Classes are more sophisticated, as they express with much more precision the relationships with other classes. C++ #includes and modules have problems, and this primitive building block method is not required in an object-oriented language.

In summary: In an object-oriented program, classes form the basis of modularity. The #include mechanism is old, and must be explicitly stated by the programmer, and is not necessary given all dependency information is implicitly derivable from the text of an object-oriented program. Multiple inheritance is a form of #include. The class interface is the object-oriented equivalent of the module header, so headers and #include mechanisms seem out of place in an object-oriented language.

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 leads to a tiring activity, and a greater chance of the eye missing the sought identifier, so that the programmer must resort to using the search function of a text editor to help out.

Other languages have chosen a different approach, placing the entities name first. For example:

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

This seems backwards at first to those used to the ALGOL and FORTRAN styles of type first. But there is a very good simple reason for it. An example from the real world may illustrate the argument for name first. Imagine if a dictionary was published, and the keywords were not placed first, but rather the entry order was -

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

the dictionary would not sell many copies, unless the marketers managed to convince large numbers of the population that there was something intrinsically magical about this order of layout, that made the explanation of the meaning more correct. This example illustrates how subtle syntax decisions can be very important, and why the PASCAL style of languages may have ordered things this way around, contrary to FORTRAN, ALGOL and others. These are trivial but important alternatives that the language designer must consider.

In summary: layout of programming entities is essential for effective communication. Layout is provided by the dual roles of language syntax, and programming style. A dictionary or index style layout would suggest placing entity names first, followed by their definition.

Calls on references to deallocated objects (Dangling Pointers)

The compromise that C++ does not have garbage-collection means that it is possible that a routine in a deallocated object may be called. For example if an object is deallocated in one place in the program, but another part of the program still has a reference to this object, then the potential to invalidly invoke a

routine exists. There is no way to detect that the object has been deallocated, as the reference will not be NULL, but a ‘dangling reference’. This contributes to the fragility of C++ programs.

The second problem is that objects which become useless may miss being deallocated. This leads to memory leaks, which means memory gradually becomes full of dead objects, leading to eventual system failure. Such memory leaks are difficult to detect and find. Such leaks are often only detected after long and thorough testing of the system.

Both problems are solved by garbage-collection. Garbage-collection has undeservedly earned a bad reputation due to some early implementations exhibiting some performance problems, instead of working transparently in the background, as they can and should. As C++ does not include garbage-collection, the problems are often over-emphasised as a justification for C++ not including it.

It may be argued that the lack of garbage-collection in C++ is not an engineering compromise, as its inclusion would be an engineering impossibility, as there are many ways that a programmer can undermine the structures that are required to implement correctly working garbage-collection. While garbage-collection may not actually be an engineering impossibility in C++ (EC++), it is difficult, and programmers would have to settle for a more restricted way of doing things. This may be a good thing, but then the compromise to remain compatible with the C language becomes difficult, if compiler detection of practices inconsistent with the correct operation of garbage-collection is required.

Type-safe linkage

As explained in the C++ ARM, type-safe linkage is not 100% type safe. If it is not 100% type-safe, then it is not safe. It is always the subtle errors that provide the most problems, not the simple or obvious ones. Such subtle errors get into the production system, undetected, until a maybe critical moment. The seriousness of this situation cannot be underestimated. As many forms of transport, such as planes, and space programs depend on software to provide safety in their operation, lives can depend on software. Not only this, but often the financial survival of organisations can depend upon their software. Therefore the acceptance of such unsafe situations is at best irresponsible.

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, and 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 in this manner. Building systems out of preexisting elements is the common Unix style of software production. This implements reusability of a form, but not in the truly flexible manner of object-oriented reusability.

This is why in the future, Unix may be replaced by completely object-oriented operating systems, which are indeed ‘open’, to be able 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 some sort of 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 not acceptable for widespread and expensive software, or safety critical applications. Since C++ is becoming widely used, this sort of implementation should not be

regarded as acceptable. For C++ implementations that do not have their own specialised linker, optimisations such as the replacement of virtual calls with procedure calls may not be possible.

C++ and the software lifecycle

Much work has been done on defining the software lifecycle. 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, one being performed and 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 quite incorrect. Someone who sits down and starts writing program text straight away is actually doing all the steps in parallel. It may not be the best way do do things in many circumstances, may or may not suit the style and thinking of different people, but there are scenarios where this works, and can be the methodology of choice of disciplined thinkers. More likely though, the phases will interact and overlap. Facts found out only as late as the implementation stage may be fed back into the analysis and design stages.

It is now becoming accepted that the software lifecycle should be an integrated process, that each phase will lead to the next naturally, without unnecessary artificial separation, and that the activities can progress in parallel, expediting software development. The object-oriented approach supports this model well. When the steps are artificially separated, this 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. What is required in the software industry is a more rational approach. We should have learnt from the extremes SA/SD. This methodology was taken to mean in some quarters that the methodology was all important, while programming and programming languages were not at all important. This attitude was strengthened by programming languages being arcane and machine-oriented. A modern software language will support the integration of the activities of design and implementation by being readable, and problem-oriented. A modern language needs to be as close to the design as possible, as the needs and requirements of an enterprise can change much more rapidly that the 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 which was designed mainly as an implementation and machine-oriented language. It is an old language, which did not need to consider the integrated lifecycle approach. C++ may 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, as the semantic gap between design languages and the implementation language is great.

We should have learnt from the structured world that this is not the correct approach to the software lifecycle. However, in the OO world we are again falling into the trap of artificially dividing up the lifecycle into distinct activities of OOA, OOD and OOP, instead of adopting an integrated approach.

In summary: C++ provides limited support for design. Modern languages will provide a much more integrated approach to the complete software development process. C++ includes some of the structures

from object-oriented programming, but does not address the entire software lifecycle, as a modern object-oriented language can.

Reusability and Communication

Reusability is a matter of communication. In order to use a software component, you must be able to understand it. To do so, the writer must communicate the purpose and intent of the software to the client, and the correct usage of that software. In the object-oriented world, clear and concise definition of software modules is therefore not a mere nicety, but essential if reusability is to be realised. Arising out of the issue of reusability is extendibility. In order to maximise the reuse of software, it will often need to be tailored to suit the new application at hand. The client programmer in this situation must decide if the software component is indeed suitable for the task in hand, and if so, what is the best way to extend it.

Reusability is a matter of Trust

Reusability is a matter of trust. If you do not have confidence in a software component, then it will be difficult to reuse. Matters for doubt may be, that it does not provide enough functionality, or even correct functionality, or that it may not be efficient enough for the purpose, or worse still may crash. If the software component cannot be trusted, then it is not a potential candidate for reuse. Unfortunately, the C/C++ philosophy that checks are not built into the language and compiler, because programmers can be trusted works against reusability. If you are a potential client of a software component, then you need to be reassured that the component is as trustworthy as possible.

Programmers make mistakes, and if you have the uncertainty that certain classes of mistakes have not been caught by the compilation system, then confidence in reusability is undermined. The C/C++ philosophy of trusting programmers is contrary to the object-oriented goal of reusability. In the real world of reusability, the furry headed ideal of trusting programmers is not appropriate. 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 suppliers programmers. The C philosophy of trusting programmers is against the commercial interest of both parties.

Concurrent Programming

In the next ten years we will probably be introducing into common use multiple processor arrays, which will execute concurrent programs. This requires much cleaner languages, than the single processor languages of today. Object-oriented concepts support concurrent programming, as objects may 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. Object-oriented programming provides a scheme to cleanly divide state spaces. In traditional sequential programming practice, the demand that everything be broken down into loosely coupled modules, that only interact through well defined interfaces may seem inefficient. However, it is precisely this scheme that will mean that concurrent solutions may be developed efficiently and transparently to the programmer.

In concurrent programming, a thread in a unit of sequential execution. Concurrency is achieved by the splitting of threads. Threads may be split when a state changing routine is invoked, but not a value returning function, as the value must be waited on. State changing routines may easily be invoked on another processor. Object level granularity seems to be a natural candidate for concurrent processing. Only one thread may be active in a single object at any time to avoid simultaneous updates. 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 irrespective of whether the customer is running a single processor, or a processor array. Concurrency is problematic where there is a global environment. C++ does not preclude the use of a global environment, and access to shared global data potentially causes a thread to lock, and where there are many such accesses, the advantage of concurrency is lost. Even the introduction of static entities does not help a great deal, as these still have the problems of sharing associated with concurrent programming. It may well prove to be the case that adapting C++ to a concurrent environment proves to be more technically difficult than the inclusion of garbage collection.

In summary: Concurrent programming will require languages which purely express the requirements of the systems being implemented. Concurrent programming will require the simplest and most consistent programming paradigms, and extremely disciplined modularisation, so that a large state space can be updated concurrently. The division of state space into objects as in OOP is essential if concurrency is to be manageable, and used to the greatest benefit. Programmers will be able to produce more powerful systems by the acceptance and adoption of more ‘restricted’ practices. It is up to the compilers and target architecture to determine the optimal way to produce an executable system.

The role of Language

For a light intermission between sections, there are some interesting points made by the Cambridge Encyclopedia of Language. 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.

Now while natural language may be difficult to be “systematic and objective about”, does this apply to computer languages? While the definition of natural language is beyond our control, programming language definition is certainly within our control. Programming languages must both have the expressiveness of natural language, yet be precise and semantically consistent. As programming languages have rigorous requirements, we should be even more critical and objective about them. Unfortunately, it is a measure of immaturity in the programming profession that criticism is often met by emotional and irrational argument. This leads many to conclude that the choice of a programming language is no more than a religious issue. This is incorrect as the choice of a good programming language should be done on technical merit, and there are technical measures by which a language may be judged effective or not. If

language choice is merely a religious issue, then we may as well still be programming in assembler, or maybe even binary. Understanding the role of language will help quantify what it is we have to measure.

The 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. Other functions it lists are: emotional expression, for instance, when we stub our toe, we often emit words, even when there is no one to hear; social interaction, for example if someone sneezes, we often “bless” them; the power of sound, as in poetry and rhyming jingles etc; the control of reality, as in spells and incantations, perhaps computer programs and spells are very close in purpose; recording the facts, record keeping, historical and geographical documents, etc; the instrument of thought, we quite often reason about things to ourselves in language; the expression of identity, it can tell the world who we are, or affirm our belonging to certain groups. Perhaps the most important role of computer languages is that of description.

“Language shapes the way we think, and determines what we can think about.” - B.L.Whorf. As mentioned before, this is quoted by Bjarne Stroustrup. But how true is this? 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.

Indeed it seems that there can be both verbal and non-verbal thought. For example following a road map in a car. But when we try to put this into words, we find it quite a tedious task. The above Whorf quote is a statement of the position of the Sapir-Whorf hypothesis on language and thought. This was formulated by Edward Sapir (1884-1939) and his pupil Benjamin Lee Whorf (1897-1941). It reflects the view of the day that 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.

The Sapir-Whorf hypothesis in its strongest form, as in the Whorf quote, is not generally accepted now. For one reason, it is known that concepts can be translated from one language into another, even if in one language, the concept may be expressed in one word, but takes a longer explanation in another. A weaker version of the Sapir-Whorf hypothesis is accepted. That is that “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 suggests that it is the concept behind the languages that are important, not the languages themselves, as they only provide a framework for the expression of the concepts. A language can only be as good as the concepts it implements, but a language must also be a simple and clean implementation of the concepts, perhaps expressing the concepts in the

equivalent of as few words and constructs as possible. Programmers who understand the concepts should have no difficulty in adapting to different languages, as long as the concepts are implemented elegantly in the new language.

A language may be judged like a wine connoisseur judges wine, by holding it up to the light to judge it 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. So with programming languages, the clarity of the definition of the language will help that which is important, producing quality software using that language.

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 are learnt, object-oriented programming seems a most natural way to program, as it matches most effectively the way we think. If C++ has been designed according to the Sapir-Whorf hypothesis, its philosophical basis is quite at odds with a computer industry that should shape tools best suited to its purposes, processes, thinking, and concepts.

Quotes taken from “The Cambridge Encyclopedia of Language”, David Crystal, (Cambridge University Press 1987).

Generic C criticisms

These criticisms apply to the C base language, but in general adversely affect C++.

Pointers

The C pointer mechanism requires the programmer to be concerned with issues that the compiler can automatically take care of. Firstly, the programmer must be concerned with the underlying address mechanism. Secondly, the programmer must be concerned with the correct dereferencing of pointers to access the referenced entity. Many languages make both concerns transparent to the programmer, being naturally the responsibility of the compiler to generate such code, with no loss of generality, or efficiency.

An example is that the C programmer has to worry about the means of parameter passing, and must work out the correct use of &s and *s. If this is incorrect, at best a syntax error may be produced, at worst incorrect code will pass into testing and production. This information is specified in the parameter declaration, so that the programmer needs no longer worry about the access mechanism in the body of the routine. This again enables flexibility, as the use of the parameter is implementation independent. The pointer mechanism of C means the programmer must constantly be aware of the implementation mechanism. This is against the principle of OOP that all data access should be achieved in an implementation independent manner. C++ rectifies this to some extent by the introduction of reference variables (is this an admission that the other languages were correct all the time?) This is another syntactic variation that programmers must be aware of.

Another small problem is the confusion caused by the style of declaration -

```
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.

Another consideration is that the mechanism for implementing dynamic memory space may change from one target environment to another. For example on the Macintosh, relocation of objects is implemented by a double indirection mechanism (handles). This is similar to the Unisys A Series mainframe approach of having all object descriptors access the target object via a master descriptor, which stores the actual address of the object. However, the A Series makes this completely transparent to programmers in all languages. Different implementations may not allow relocation at all, and thus the need for double indirection imposes an unnecessary performance penalty. Such details should not be the concern of the programmer, but rather the compiler in the target environment.

The manipulation of C pointers is error prone. A pointer may be incremented past the end of the entities they are pointing at, with subsequent updates via the pointer possibly corrupting other entities. How many

lurking and undetected errors may be in programs because of this? This again illustrates how C undermines OOP by providing a mechanism where state outside of an object's boundaries may be changed. This problem is exacerbated by the fact that pointers are intrinsic to the method of writing software in C, and the techniques are so widely used that many programmers are unable to understand how software can be produced without their use. Pointers as implemented in C make the introduction of advanced concepts like garbage collection and concurrency difficult.

Arrays

As noted on p 137 of the C++ ARM, C arrays are low level, and yet not very general, and unsafe. Modern software production has far less dependence 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 seriously undermines the semantics of an array declaration, ie that an array is declared to be of a particular size, and can only be indexed by values within the given bounds. If also compromises software safety. Also, in C there is no notion of dynamically allocated arrays, whose bounds are determined at run time, as in ALGOL 60. This limits the flexibility of arrays. Thus the C definition of arrays compromises both program safety and flexibility.

Arrays are in general just another object-oriented entity, and should be treated in an object-oriented manner as a class of data structure, to which are applied the interface definitions, and consistency checks inherent in object-oriented systems.

D.C. Ince in ACM SIGPLAN Notices, January 1992, in an article "Arrays and Pointers Considered Harmful", explains why 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 may 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. Perhaps even more so should C and C++ be regarded, as pointers and arrays are far more intrinsic in the use of C and C++.

Function Parameters

One use of pointers is to simulate by-reference parameters with by-value parameters. Being able to distinguish between by-value and by-reference parameters is not just a syntactic nicety, included in most high level languages, but a valuable compiler technique. It means that the compiler can detect where dereferencing is needed, and can generate code accordingly, without requiring the programmer to be concerned about this. However, most languages do not even get this correct, allowing assignment to the local copy of by-value parameters. By-value parameters should be read-only, and not allowed to be the target of an assignment within a routine. By reference parameters mean that the value of a parameter passed to a routine may change. This introduces a mechanism of updating the state space, other than straight assignment (although the routine may use assignment to achieve the 'dirty deed'.) Such value changing mechanisms introduce side-effects.

Object-oriented principles suggest that parameters should not be by-reference at all, whether given direct language support, or simulated support as in C. OOP realises that by-reference parameters are another mechanism that allow side-effects, changing the environment beyond the boundary of the current object. Parameters need only be read-only, and if a change is required to an object passed as a parameter, then it must be performed by calling a routine encapsulated in that object. This removes the possibility of the side-effects of by-reference parameters, and means that an object cannot change the environment outside of itself by assignment to a by-reference parameter within a routine. That an entity may be updated via a point external to itself, is quite opposite to the data hiding principle of object-oriented programming.

void *

"Passing paths that climb half way into the void" - Close to the Edge, Yes

Is this the programming equivalent of an oxymoron? That you can declare a pointer to void suggests some sort of semantic nonsense, a dangling pointer perhaps? While we can have some fun conjecturing what some of the obscure syntax of C++ may suggest, a serious problem is that void * declarations are used to defeat the type system, and thereby compromise the advantage of having a type system. A well thought out type system requires no such facility.

When a typed entity is assigned to a reference of void *, it looses all 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 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 may be mistakenly applied to objects of another class.

void fn ()

The default type that a function returns is int. The default should be a typeless routine, returning nothing. 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. In C though the function must be declared void. Also a typed function may be invoked independently of an expression. It is a shorthand way of discarding the value returned, which means the compiler does not detect that the function invocation has failed to match the functions signature. Thus an important type of consistency check is not made. Values should be returned because they need to communicate with the outside world, and ignoring returned values is often dangerous.

fn ()

Empty parenthesis is the invocation operator in C. This relates C to languages such as FORTRAN and COBOL, where the specific CALL and PERFORM instructions were required to indicate routine invocation. This similarity of C to FORTRAN and COBOL is not so obvious, as the () operator is more mathematical looking in nature. In the ALGOL style of languages, invocation is automatically deduced by the compiler when it sees a routine name, as it knows that the identifier refers to a routine. This compiler technology was not realised at the time FORTRAN and COBOL were developed. This compiler

technology is possible, because much information is stored by the compiler about the entity, and the compiler can check that subsequent use by the programmer is consistent with the declaration, and in many cases the compiler can be relied upon to generate correct code, without having to burden the programmer with having to redundantly supply this information. This contributes to flexibility and implementation independence.

A specific invocation operator results in an artificial distinction between functions and variables. Indeed, a variable used in an expression is a value returning function, it just returns the value stored in the variables memory location, rather than calling a function to compute the value. Functions and variables are the implementation viewpoint of the problem-oriented concept of attributes. Attributes may be both static, and dynamic. For example, a printer may have static attributes like pages per minute, printer type (laser, line, etc). A printer may also have dynamic attributes, like the number of print jobs queued, pages left in the current job, etc. However, the user of the attributes should not be concerned, whether they are static or dynamic, just that the interface returns a value of a particular type. This view also ensures implementation independence.

The important semantic information of an identifier to the programmer is that it returns a value of a given type. The programmer should not care whether the value is the result of a routine, or the value stored in a memory location. What is important is that the value is of a type compatible with the expression in which it occurs. This can be checked by the compiler.

The function invocation distinction reduces flexibility. If for optimisation, a function is replaced by a precomputed variable, it is not possible to use the same name without removing all the () from the original function invocations. This may be spread over many files, and such optimisation may not be attempted by the programmer to avoid the tedium associated with the task. Thus implementation detail is visible for the outside world to see.

If the rarer circumstance of needing a reference to a function is required, then this special situation should have the extra explicit syntax, not the common invocation situation. Having such function pointers though is analogous to the call by name facility in ALGOL, and this was recognised as having pitfalls. All these pitfalls may be avoided by consistent application of the object-oriented paradigm.

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, and the design of a program may take advantage of this fact, without resorting to explicit jump tables. Another use is to jump to a function in a table, which is indexed by an input character. This mechanism is better catered for by a switch statement, which makes what is meant explicit, while keeping underlying mechanisms (and possibly optimisations) implicit. C++ allows function pointers to member functions to be stored in tables (via the .* and ->* operators). This though cannot take advantage of dynamic dispatch through the virtual jump table.

In summary: Mathematically, any entity which returns a value of a given type is a function. This includes variables used in expressions, so distinguishing between variables and functions introduces an artificial distinction into the language. This reduces flexibility. Distinguishing between static and dynamic attributes, reduces implementation independence. The () operator is equivalent to the FORTRAN CALL, and COBOL PERFORM, only the syntax is different. How to handle an entity should be left up to the compiler, as the programmer has already specified the intended usage by the nature of the entities declaration.

++, --

These are another unnecessary shorthand convenience. The shorthand += and -= is more powerful, as the variable may be incremented by values other than 1. This illustrates that 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. If it is mistakenly believed that the other forms 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. The last two forms a++ and ++a are the postfix and prefix forms. They are often used in the context of another expression. Thus one expression may be used to perform several updates. This is a very powerful and convenient feature, but introduces side effects into an expression, which sometimes have surprising effects, and may 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
```

This example the ARM points out should be detectable by compilers, but the exact interpretation appears to be left to the implementation, which may contribute to non-portability.

It is better to program in a straightforward manner, as an optimising compiler will 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 main slow memory several times. The nature of such optimisations depend on the machines architecture, which a programmer should not be aware of, or expected to be aware of, as open systems demands that programs should be able to be ported amongst diverse architectures and environments, very different to the original machine.

As with name overloading, memory storage update is a problematic, but necessary part of programming. It should be provided by a language in a consistent and expected way. Most languages have realised that memory update is problematic, and typically only provide limited but sufficient ways of updating, by an assignment operation. (Many languages also have block memory copies as well, but block copy could also be provided by assignment.) Furthermore, most languages avoid side-effects by limiting updates to only one per statement. C provides memory update in too many ways. These ways add nothing to the generality of the language, but do add ways that program errors may be produced.

In summary: Increment and decrement operators introduce no added benefits for optimisation with respect to modern compilation environments. These can introduce side-effects by multiple updates within one expression. These operators introduce no extra generality or power of expression into the language.

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 semantic meaning of constructs.

Case Distinction

While it is good to adopt some typographic convention for the presentation of names the fact that C distinguishes between upper and lower case in names can cause confusion. Whether a character is written 'i' or 'I' it remains the same character. We would not make the distinction if the i were italicised or emboldened. Case distinction is a weak form of name overloading. If you can overload names in such a way, you can easily use the wrong one. For example if an identifier is declared Fred, another one may be declared fRed. Such names may easily be mistyped or confused. Typographic conventions should not affect the semantics of a program (consider RPG).

As every programmer will have experienced, one character errors are more difficult to detect than one would think. We are in general not good proof-readers. There is a psychological reason for this 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. In order to overcome this programmers have to 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. Modern language design takes into account such psychological considerations in these small but important details, being designed towards the way the human brain works, not towards the way computers work. Such considerations 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?

As has been suggested, case distinction provides another form of name overloading. Name overloading is a double-edged sword, as it is the attachment of different semantics to the same name. This opens up the possibility of confusion, but it is difficult to imagine programming without it, and its corresponding notion of scope. In object-oriented programming, name overloading is used in a very powerful, but disciplined and restricted manner in order to limit the problems and confusion that are inherent to name overloading. Case distinction can result in unexpected name overloading that can cause confusion and errors. Name overloading as has been suggested in the section on name overloading should only be provided in controlled and expected ways. This means that where a name is overloaded in the same scope that the compiler should report an error, or require the programmer to explicitly state that this is the intention, as with Object Pascal's override.

As another example, a commonly used technique is -

```
class obj
{
    int   Entry;

    void  set_entry (int entry)
    {
```

```
            entry = Entry;
        }
}
```

Upper and lower case distinction also illustrates another old compromise that C made. The compromise was that upper and lower case letters are represented by distinct codes in the ASCII character set, and to remove this distinction required precious computer time. Most languages overcame this problem by an even worse restriction, that everything had to be in upper case. C's mix of upper and lower case seemed good in comparison, but it was still not correct. These old languages and C considered processor time to be more important to convenience for people. But 20 years ago perhaps it was, as computers were expensive and slow. This assumption has been removed a long time ago, but we are still stuck with it in C based languages, and the Unix operating system. Early compilers also had restrictions (original definitions of C say that only the first 8 characters of identifiers were significant), on the length of identifiers to between six and ten characters, and this meant that the upper lower case distinction gave the benefits of an extended character set. This consideration no longer applies in all but the most basic compilers.

As an aside note, on a typically common practice in any language, due to historically short identifiers programmers shorten identifiers by the omission of vowels. This leads to error and ambiguity, and difficulty in reading and interpretation at a later date. For example do the characters 'prvt' represent 'private' or 'pervert'? This slows good typists down, and poor typists probably even more, while they work out exactly which vowels to miss out. This shows that many programmers have not adapted their practices as the limitations of technology have been removed.

In summary: The language designer has two alternatives. To either have case distinction or not. Case distinction can lead to rare but difficult to detect errors, due to yet another form of name overloading. The prohibition of case distinction may occasionally force the programmer to have to choose an alternative identifier, but the fact that a potentially erroneous name clash has occurred will have been detected early. The event of error from case distinction may be very rare, but then again would you trust your space shuttle to even the remotest chance of something going wrong, where billions of dollars are at risk?

Assignment Operator

Using the mathematical equality symbol for the assignment operator is an example of 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 mathematical sense. In C the lack of distinction leads to error. It is easy to use = (assignment) where == (equality) is intended, and the result looks quite reasonable, resulting in errors that are difficult to detect.

In fact this leads to a more general criticism of C, in that it has a pseudo mathematical appearance. There are very few people who are proficient at the interpretation of mathematical theorems, most passing over such sections in text, unfortunately often making the assumption that if the mathematics is there, the surrounding text has been proved correct. Generally, the pseudo-mathematical nature of C shares this bad attribute of mathematical notation as it is difficult to read, while lacking the semantic consistency and precision of mathematical notation. One of the keys to reusability is readability.

Type Casting

Type casting is just a specific form of mathematical function, which maps values of one type onto values of another type. Type casting has been useful in computer systems, as 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 cannot be implemented by more general functions. Furthermore, type casting causes problems in strongly typed systems, as it compromises the type system. In many languages, type casting is necessary, as the type system has not been consistently defined, so programmers often feel they have the need of the type casting mechanism, and may find it difficult to imagine languages where type casting is not required at all.

Mathematically, all functions can be regarded as having a type casting function. An example often used in programming is to cast between characters and integers. Type casts between integers and characters may easily be expressed mathematically 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 it must be specified exactly which subset of integers will map to characters by use of a precondition. Now this may be provided consistently in object-oriented syntax 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.
```

Now such basic data types, as character and integer, are usually catered for well by the hardware of most machines, and it is entirely possible that a compiler will generate code that is optimal for any target hardware architecture. However, such basic data types may be consistently and elegantly treated in the object-oriented paradigm, by the implicit definition of their own classes.

Another example of type conversion is from real to integer. Here though, the programmer may 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 may indeed generate the code best suited to the target machine.

In summary: Type casting compromises the advantages of type safety. Type casting in a pure object-oriented system is not needed, as it is only a specific implementation of the more general mathematical notion of functions.

Semicolons

A program is a list of elements, and the executable part of a program is a list of sequentially executed instructions. Languages such as FORTRAN separated instructions by requiring that they be placed on different lines or cards. Elements in a list need to be separated in some manner, and the semicolon is one syntax for the separation of the elements in the list. The semicolon is therefore part of the syntax of the list, not part of the syntax of the individual instructions. But C makes the semicolon part of the syntax of each instruction by making the semicolon, a terminator of individual instructions. This elevates an unimportant semicolon to have a much higher syntactic value than it should. Imagine if the comma was a terminator, then function invocations would have to look like:

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

C's handling of the grammar of semi-colons 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 both of the above will be reduced by a parser to the grammatical form:

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

Conclusions

C++ is overly complex. Object-oriented languages should provide sophisticated concepts in the simplest possible framework. Where the framework is not simple, the concepts tend to be lost, and there are many issues that OOP addresses 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++. There are many potential ways of introducing subtle errors into C++ software, and furthermore, the combination of these will cause even further problems.

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. To manage such state spaces requires the simplest techniques, in order to guard against detectable inconsistencies which may lead to errors in executable systems. C and C++ do not implement the simple management of a large state space, and also allow to 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 a consistency checks are often viewed. Programming languages should embody the collective wisdom of common sense practices, which have been learnt over many years, by common and painful experience. C++ lacks the implementation of much of this wisdom.

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 the development process down, and is therefore costly. Good programmers in this context (often labeled ‘gurus’), are those who recognise symptoms, and recommend fixes. Good programmers in the better sense (often labeled ‘impractical idealistic dreamers’) adopt better practices (programming languages being a subset of these), that avoid error in the first place.

The view that correctness checks are training wheels for students needs to 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 a teaching aid, but as a tool that helps produce a polished and professional performance.

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. It is therefore suggested that the move to pure object-oriented languages will indeed facilitate more consistent programming, which will avoid many errors which typically occur in software production. 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. In chopping down a tree, if the axe is blunt, then procedures, processes and methodologies may be invented to make the blunt axe as effective as possible. However, that does not solve the real problem that the axe that does the real work is blunt. So it is with programming languages. If a system is to be realised, then it must be implemented, and a programming language is the tool with which the real work is done. If the language is blunt, then procedures, processes and methodologies may alleviate that situation, but they do not solve the cause of the problem. Once the axe is sharpened, then real progress may be 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 may not be effective with

even a sharp axe, but that does not mean that the axe maker will not bother 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.

The critique began with certain questions, and as no work can be absolute (particularly a programming language), it will end with more questions, which 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 by 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 language 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
April 1992