

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

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

3.21 Optional Parameters

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

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

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

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

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

Other means, already in the language, can easily provide this mechanism. For example, a call to another (possibly inline) function could provide the defaults for the optional parameters:

```
g(a, b, e);           // the call
g(int a, b, e)       // the function
  {f(a, b, 0, 0, e);}
```

This not only provides the convenience of optional parameters, but is more powerful. Any parameter or combination can be filled in with any combination of defaults, not just the last parameters. Multiple intermediate routines can provide multiple sets of defaults.

Neither Java nor Eiffel have optional parameters. Strong typing is enforced, so that the parameters of a call must match the routine signature.

3.22 Bad Deletions

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

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

One of the restrictions of the design of C++ is that it must remain compatible with C. This results in examples like the above, that are ill-defined language constructs, that can only be covered by warnings of potential disaster. Removal of such language deficiencies would result in loss of compatibility with C. This might be a good thing if problems such as the above disappear. But then the resultant language might be so far removed from C that C might be best abandoned altogether.

Bad deletions are the kind of problem the Java designers set out to avoid. You do not get bad deletions in either Java or Eiffel for two reasons: firstly, they do not have pointers; secondly, they provide garbage collection so don’t delete objects.

3.23 Local entity declarations

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

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

In well written object-oriented software, routines will be small, typically performing one atomic operation per routine, so localised declarations will not be of much value. Small routines that implement atomic operations are fundamental to loose coupling. For example, a base class that provides a single routine that logically performs operations A and B, is not useful to a subclass that needs to provide its own implementation of B, but does not want to change A: the descendant must reimplement the logic of both A and B, missing an opportunity to reuse the logic of A. Splitting A and B into different routines accomplishes loose coupling, and therefore flexibility. Tight coupling reduces flexibility.

Efficiency is also attained without the mess of local entity declarations. Good design and clean modularisation achieve efficiency, as the entities

which would be locals to a block in C++ are only created when the routine is entered. Furthermore small routines can be inlined, and in this case, the locals will only be created when the expanded inline block is entered, which is the same effect as if the programmer had included the block manually.

Java implements locals in the same way as C++. In Eiffel the philosophy is to use good design to make routines sufficiently small and atomic. That is one operation, one routine. With this approach, having local declarations only in one place in the routine and not throughout is sufficient. If you find a place where you want to introduce local variables within the code, this is an indication that you should write it as a separate routine. An objection could be that small routines with lots of overhead calling them is not efficient. Eiffel compilers solve this by automatically inlining routines. Thus the integrity of a design is preserved in the program text, but efficiency is retained. In C++ you could manually inline such functions.

3.24 Members

Care should be taken with the C++ use of the term member. In general use, an object is a member of a class. For example, squirrels are a member of the class animal. This corresponds to members in set theory. But in C++, the term member means a data item, or function of the class. Some people might say that set theory is one thing, but programming is another, so there is no problem with using the terminology. However, set theory underpins the theory of computation and programming, and sets, classes and types are related. Sets are a means of describing groups of entities which have some similarity. Supersets group entities according to broad concepts; subsets group entities according to narrower concepts, that is, more restrictive criteria. So sets also underpin our understanding of classes and subclasses.

In set theory we say: $3 \in \mathbb{N}$, or 3 is a member of the set of natural numbers. In objects we would say that Fred is a member of the class person. In C++ the field name which for some object contains the string "Fred" is a member of the class person. This is not mathematically correct, and the confusion could have been avoided.

Java does not seem to use the term member. It might stick from C++. Eiffel uses the term *features*.

3.25 Inlines

The problems described in this section are a consequence of placing the burden of encapsulation on the programmer. You might wish to review the section on encapsulation at this point.

The main reason inlines were introduced in C++ was to alleviate the cost of crossing the 'protection barrier', [Stroustrup 94]. The protection barrier in C++ is data hiding. When accessing a data item in C++, it is recommended not to do it directly, but via a class member function. For example, given an

object reference *c* you should not access the data member *di* directly:

```
i = c.di; // Not recommended C++ style.
```

instead *di* should be private and accessed as follows:

```
i = c.get_di();
```

where *get_di* is:

```
int C::get_di() {return di;}
```

However, Stroustrup found that some programmers were not using an access function because of the overhead of a function call. So inlines were introduced:

```
inline int C::get_di() {return di;}
```

Note that this style of data hiding clutters the name space and text of a class.

The inline mechanism has two conceptual mistakes and a practical one. Firstly, data hiding and implementation hiding are not the same. Implementation hiding is more to do with hiding the mechanics of the access mechanism, so that you can't tell whether it is a constant, variable of function you are accessing. Inlines are the wrong solution to this problem: the correct solution is uniform access. The OO concept is to hide implementation; data need not be private, but may be functionally exported from the classes interface.

This leads to the second conceptual mistake that functional access and C functions are different things. Functional access hides the access mechanism. C functions, however, make the access mechanism visible: you know you are invoking a piece of code that will be jumped to. Functional access by contrast is any entity name that can occur in the context of an expression. This entity could be a constant, variable or value returning routine, but you can't tell which if the implementation of the access mechanism is hidden. The statement *i = c.di* is functional access. C++ has solved this problem in exactly the wrong way in order to stay compatible with the flawed concept of function in C.

The programmer is required to bear this burden, which in turn makes software development more costly for every company using C++, and again flexibility is reduced. In order to restore information hiding, that is access transparency between constants, variables and C functions, programmers must as a matter of style hide constants and variables behind a C function, as is the case with *get_di()*. A fix to the language would have been better, but not possible to keep compatibility with C.

The practical mistake is that a compiler can automatically generate inlines. Requiring a programmer to specify *inline* is a manual bookkeeping task. It is not hard for a compiler or optimiser to work out that *C::get_di()* {*return di;*} or even more complex routines could be inlined. This is exactly the kind of

optimisation that Eiffel and other sophisticated languages perform.

[Flan 96] says: “A good Java compiler should automatically be able to “inline” short Java methods where appropriate.” An article in Byte of September 1996 suggests that to optimise Java method calls, “you should make liberal use of the `final` keyword.” Byte also suggests that instead of small functions, programmers should inline by hand small methods. Byte further says: “The trade-off, then, is either better performance or code flexibility. You must decide which is most important to the program’s operation in that situation.”

In this respect, Eiffel again proves itself superior. Eiffel automatically determines that a routine is `final`, or in C++’s terminology, that a routine is not `virtual`. Also Eiffel automatically inlines. Therefore the Eiffel programmer does not need to bend the code to gain performance, or consider trade-offs: you do not have to trade-off flexibility to gain performance.

Eiffel has a further advantage that it understands the difference between implementation hiding and data hiding and provides implementation hiding. It also accesses data and constants functionally, so in the instruction:

```
i := c.di
```

you can’t tell and don’t need to know whether *di* is implemented as a constant, variable or routine function. The implementation is hidden: access is uniform as access to a constant or variable looks the same as a value returning routine, and the different access mechanisms behind these is hidden and automatically generated by the compiler. And since this implementation distinction is hidden, the need is greatly reduced for either the programmer to manually inline, or for the compiler to automatically inline. In this case Eiffel provides the maximum flexibility.

Since C functions are poor cousins to mathematical functions, and C++ also confuses data hiding and implementation hiding, the language includes otherwise unnecessary mechanisms like inline.

3.26 Friends

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

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

Friends are useful, and a case can be made for shades of grey between public, protected and private

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

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

As mentioned in the section on inlines, implementation hiding is different to data hiding. As long as you access your data functionally, you do not have to hide your data, just the access mechanism.

Another question is, since there are inlines, is there a need for the similar mechanism of friends? If you mark a function inline, it is going to expand inline, and avoid the function call overhead. So in this case, friend is a superfluous mechanism.

In Java, classes in the same package can access instance variables from other classes in a *friendly* fashion. This is contrary to good programming practice and OO design, as it means you can access things without going through the published interface of a class. However, in Java, explicit friends are gone.

Eiffel offers the pure OO approach, where everything must go through publicised interfaces. Note in Eiffel that data attributes in a class may be exported in the published interface, as access is uniform. In that case, external entities can read the data, as if it were invoking a function, but you cannot write to a data item in an external class. To update a data item, you must call an update procedure. Part of the purpose of friend is to update an item directly, without the overhead of a procedure call. In Eiffel the compiler will automatically inline procedures where possible, so the efficiency concern is addressed.

To summarise: Eiffel does not need the friend mechanism for two reasons: firstly, external classes can access data attributes for reading; secondly, for update, a procedure is expanded inline where practical. Accessing a data item does not contravene encapsulation or implementation hiding. Data hiding is not encapsulation, although with encapsulation *implementation* data is hidden, the operative word being ‘implementation’, not ‘data’.

3.27 Controlled exports vs friends

As noted in the section on friends, there is a case for finer grained control of exports than `public`, `private` and `protected`. Except for friends, Java uses the same mechanism as C++, but adds two more categories, `default` and `private`

protected. This complicates the mechanism, and it is difficult to remember exactly what each category does. Eiffel does not have friends, it allows classes to be related by a finer grained export mechanism; for any set of features, you can specify exactly what classes they are exported to. Classes that are closely related export to each other interfaces that are not available to other classes outside of that group.

Also in Eiffel, you can export a routine to a different set of classes based on whether it is called as a creation routine (constructor), or normal routine call.

In Eiffel all features are implicitly `public`. Public can also be explicitly stated by exporting to class *ANY*, ie., the universal set. If a set of features is to be `protected`, ie., internal and not visible to clients, it is exported to class *NONE*. Such a set of features is *secret*. *NONE* is the equivalent of the empty set in set theory, which is notionally a subset of all sets and *NONE* is a subclass of all other classes, and has only one possible value: *Void*.

There is no equivalent of `private` in Eiffel, where features can be hidden from sub-classes. But this is not necessary, and in most cases `private` is undesirable. The Eiffel philosophy is that with inheritance you get unrestricted access to the implementation as this is key to the flexibility of reuse and extension. As a subclass, you can redefine any routine inherited from a parent. When you redefine a routine, you are changing the implementation. Since you are changing the implementation, the `private` restriction could be a nuisance to some subclass that hasn't been written yet. If you need to access a variable, and the parent class designer has made it `private`, you are out of luck. At the best you could go to the programmer who owns that class, and try to convince them to make the variable `protected`. Good luck: that kind of request often generates a lot of heat. At the worst you can do nothing about it because the class might be from outside and closed to you. Again in C++ the parent class designer is forced to make decisions that should be left open. I would recommend against using `private`, use `protected` instead. At least `protected` leaves the class open under inheritance.

In C++, `private` only restricts access, it does not restrict visibility in a subclass. With `private`, it is still possible to redefine a `private` virtual function from a base class in a subclass. This is not a problem, but you cannot prevent redefinition in a subclass, as you can with the Eiffel **frozen** mechanism.

In Java you cannot override a `private` method, but you can overload it: "Note that a `private` method is never accessible to subclasses and so cannot be hidden or overridden in the technical sense of those terms. This means that a subclass can declare a method with the same signature as a `private` method in one of its superclasses, and there is no requirement that the return type or throws clause of

such a method bear any relationship to those of the `private` method in the superclass." [Sun 96].

A further complication in C++ is that `public`, `private`, `protected` can be specified when inheriting a base class. This gives one policy for how every inherited member from the base class is to be treated in the new class. A problem with this is that once a member is `private` or `protected`, it cannot be reexported, ie., `protected` cannot be made `public`, and `private` cannot be made `protected` or `public`. Thus the temptation for a C++ programmer is to keep things `public`, as a derived class might want something to be `public`, even though it does not make sense to be `public` in the base class. Again decisions must be made early on issues you don't know about.

Java has no equivalent. Each member is inherited with the same `public`, `private`, `protected` attribute as the base class.

Eiffel again has a more fine grained approach. The export policy for each feature inherited from a parent class can be reviewed on a case by case basis. The export status of each feature can be changed and made more or less restrictive. If there is no new export policy, the default is the same as the parent class. The designer of a parent class does not have to consider what descendant classes need, or worry about the case where their needs will be in conflict with each other, as the designer of the descendant class has complete flexibility, which enhances reuse and extensibility. Eiffel's export mechanism is therefore vastly superior to the C++ approach.

3.28 Static

The word 'static' is confusing in C++. Page 98 of the C++ Annotated Reference Manual (ARM) mentions this confusion and gives two meanings: a class can have static members, and a function can have static entities; and the second meaning comes from C, where a static entity is local in scope to the current file. The choice of different keywords would easily solve this confusing use of the same keyword for several meanings. There is also a third more general meaning that objects are statically or automatically allocated and deallocated on the stack when a block is entered and exited, as opposed to dynamically allocated in free space. Another general use of the word 'static' is in 'static type checking', which obviously has no relation to the C uses, but overloads the language even further.

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

Locals declared in functions can also be static. These are not needed in an object-oriented language. The reason and history is this: ALGOL has the notion of 'OWN' locals in blocks. The semantics of an OWN entity is that when a block is exited, the value of the OWN is preserved for the next entry to the block, ie., the value is persistent. The

implementation is that at compile time, the OWN entity is limited in scope to the block, but at run time, it is located in the global stack frame. The same instance of the variable is used in all invocations of the procedure, rather than each invocation using separate local storage on the stack. This causes complication in recursion.

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

Java implements class variables with static. Eiffel uses **once** routines in order to do away with globals.

3.29 Union

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

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

Union can also be used to suppress type checking. [Stroustrup 94] says "programmers should know that unions and unchecked function arguments are inherently dangerous, should be avoided whenever possible, and should be handled with special care when actually needed."

Sun recognises that the union construct is unnecessary, and has removed it from Java. No equivalent exists in Eiffel.

3.30 Structs

Struct is only in C++ as a compatibility mechanism to C. When you have classes you don't need structs. Again, C++ is unnecessarily complicated with unneeded features.

[Sun 95] says: "The Java language has no structures or unions as complex data types. You don't need structures and unions when you have classes - you can achieve the same effect simply by using instance variables of a class."

Eiffel and Smalltalk similarly have no equivalents to struct.

3.31 Typedefs

Typedef is yet another mechanism not needed. Java, Eiffel and Smalltalk all build their type mechanisms around classes.

3.32 Namespaces

Namespaces are a new concept introduced in July 1993. Namespaces address the problem that global names imported from different .h header files can clash. The C++ solution is namespaces, where globals are put in a namespace. Access to these entities must be qualified with the namespace name. For example, A::x means access entity x in namespace A. Another namespace B might also have an entity named x, but these names will not clash. Entities not in a namespace are considered to be in the *global* namespace.

In pure OO languages, namespaces are not needed; classes themselves are namespaces. There are no global environments, so C++ introduces complexities not needed in Java, Eiffel and Smalltalk.

Java and Smalltalk have class variables, which can be used in place of globals. Eiffel provides **once** routines, so that you can access object instances where your 'globals' are stored.

Namespaces address the problem of name clashing entities. However, the names of the namespaces themselves can clash. For example, if two header files have namespaces called MY_NS, you have a clash.

As you might be aware by now, name clashes are a nuisance whenever you mix and match software entities together. An example we have seen is multiple inheritance. Eiffel provides a good solution to this with the rename clause in the inheritance clause.

Eiffel could also have a problem with class name clashes, as class names are global. The solution to this is to use a deployment language separate from Eiffel itself. This language is called LACE, Language for the Assembly of Classes in Eiffel. The concern of LACE is to mix and match class libraries together, and it provides mechanisms to rename classes, and resolve other conflicts. That way, deployment concerns are kept separate from the programming concerns.

While namespaces in C++ address a problem, they rely on programmers to be courteous, and place globals in namespaces. Perhaps a better way, would be to have a separate mechanism equivalent to Eiffel's LACE where such conflicts are resolved, rather than making the language even more complex.

3.33 Header Files

In C++ a class interface must be maintained separately from its body. An abstract class interface is just the class with the implementation detail removed so the interface and implementation can both be maintained in one source. In C++ though, programmers must maintain the two sets of information. This is because of the C/Unix style of programming with separate modules but little or no global analysis. Replicated information has the well known drawback that in the event of change, both copies must be updated. Sun calls this "The Fragile Superclass Problem." [Sun 95] This can lead to inconsistencies that must be detected and corrected. Classes that depend on another class must be recompiled if the layout of that class changes. Tools can automatically extract abstract class descriptions from class implementations, and guarantee consistency.

Splitting C and C++ programs into a myriad of small, separately compiled files turns out not to be a good way to organise projects, and not a good way to program, as you must maintain many header files. Some people are now finding it more convenient to keep an entire large system in one file as it solves many maintenance problems, and also makes it easier to find things during editing. Unfortunately, while this scheme on many systems allows for global analysis, this will still not solve the problems arising from lack of global analysis in C++.

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

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

```
#ifndef thismod
#define thismod
```

```
... rest of header
#endif
```

Headers show how C++ addresses the problem of independent modules with a non-object-oriented approach that is sub-optimal; the programmer must supply this bookkeeping information manually. `#include` relates to the organisation and administration of a project. Rational language design eliminates such manual bookkeeping mechanisms.

A class interface is equivalent to a module header. A module header contains data and routines exported to other modules. This is exactly the purpose of the class interface. Furthermore, in C++ a tool like `make` must be used to specify the dependencies.

A class definition contains all knowledge of accessed classes and their dependencies (inheritance and client) in the class text. Dependency analysis is derivable from the class text, and much of the functionality of tools like `make` can be integrated into the compiler, so the errors and tedium encountered in the use of `make` are avoided. Dependency analysis also implements a level of *dead code elimination*.

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

According to Stroustrup C++ would be a better language without the C preprocessor. Most uses of `#define` are now covered by other mechanisms. To remove `#include` would require some other import mechanism. [Stroustrup 94] says: "I'd like to see Cpp abolished."

Neither Java, nor Eiffel need header files or the `#include` mechanism. This means that programmers do not have to maintain headers separately. When Eiffel sees any declaration:

```
c: C
```

it knows the current class has a dependency on the class *C*. *C* is implicitly imported, so there is no `#include` mechanism: Eiffel has done the dependency analysis for you. If you add a new declaration to a class that hasn't be used before, the dependency is automatically generated the next time the class is compiled.

Java maps qualified class names such as `java.lang.Math` to the environments file directory structure, for example `java/lang/Math` in Unix.

Eiffel provides a utility *short* that extracts class interface definitions from the class implementation. However, the function of this is for human readability, not to provide the compiler with class definitions as in a C header file.

Eiffel also separates the bookkeeping concerns from the language. These functions are provided by

the *LACE* language, *Language for the Assembly of Classes in Eiffel*. LACE is used separately to the Eiffel language, but is processed by the compiler to map class names to their location (directory and file name in Unix style systems.)

Java and Eiffel also remove the need for *make*. Gone is the manual dependency analysis, or remembering to rerun *makemake*, when your dependencies change.

3.34 Class Interfaces

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

3.35 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 and modifiers, which is a tiring activity. There is a greater chance of missing the sought identifier, and the programmer must resort to using the search function of a text editor to help out.

Other languages place the entity names first. For example:

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

To those used to the ALGOL and FORTRAN style of type first, this seems backwards. But name first is

logical as a real world example illustrates: imagine if a dictionary was published where the keywords were not placed first, but rather the entry order is -

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

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

Java obviously has to retain this problem since it is C based. In fact the *hello world* program in Java shows how putting an entity name after modifiers can obscure the program:

```
public static void main(...)
```

Eiffel mostly puts the feature name first, except for the **frozen** case, so that features are easier to find. The **frozen** modifier is not used very often though.

3.36 Garbage Collection

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

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

Dangling Correct Memory
Pointers ↔ System ↔ Leaks

These problems contribute to the fragility of C++ programs, and usually result in system failure. Garbage-collection solves both problems, but has an undeserved bad reputation due to some early garbage-collectors having performance problems,

instead of working transparently in the background, as they can and should. These problems are often over-emphasised as a justification for C++ ignoring garbage collection. A possible solution is to build garbage collection into the run-time architecture, but allow the programmer to activate and deactivate it manually. Garbage collection can be disabled in systems where it is inappropriate.

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

[Sun 95] states that “explicit memory management has proved to be a fruitful source of bugs, crashes, memory leaks and poor performance.” Sun have built garbage collection into Java.

Bertrand Meyer lists garbage collection in his steps to object-oriented happiness. This is not surprising in a language that has exception handling, keeping track of live and dead objects is even more difficult, so Eiffel is also based on built-in garbage collection.

Stroustrup is also an advocate of optional garbage collection. In [Stroustrup 94] he states “When (not if) garbage collection becomes available, we will have two ways of writing C++ programs.” My question is not if or when, but how? Unless you restrict pointers and pointer operations, garbage collection will be very difficult, and probably inefficient. By inefficient, I mean either slow, or it won't clean up very well, or even both.

In Eiffel garbage collection is also optional. The garbage collector can be disabled during critical real time phases of program execution. It cannot be completely disabled, as if a program runs out of memory in this state, the garbage collector will be invoked, which is always preferable to the application crashing irrecoverably.

3.37 Low level coding

One of the stated advantages of C++ is that you can get free and easy access to machine level details. This comes with a down side: if you make a great deal of use of low level coding your programs will not be economically portable.

Java has removed all of this from C, and one of Java's great strengths is its portability between systems, even without recompilation.

The Eiffel solution is somewhat different again. In Eiffel you have no access to machine and

environment level details, in the language itself. You can use libraries that provide access to routines written in external languages like C. You can still write your low level C routines, and easily access this level from Eiffel. The major advantage of this approach is that all system level code is centralised in a few places, and this provides good *separation of concerns*. If you have to port your system, you know exactly which parts of code will need attention. System interfaces are thus provided in a set of well designed classes and routines. In C++ you can only enforce this as a matter of discipline over your programmers.

3.38 Signature Variance

When redefining a routine, there is an opportunity to redefine the signature as well. There are three ways a language can do this known as: no variance, contra-variance, and co-variance. This is an issue of type safety.

No variance means that the language does not permit the signature to change. The signature must exactly match the signature inherited from the parent.

Contra-variance means that the signature in a subclass can modify each argument so that it is a superclass of the matching parent argument. For example, if you have classes A and B, and B inherits from A, then given a parameter of type B in your parent, you can keep it as B or modify it to A. This does seem counter intuitive, but there are some good examples of where it works.

Co-variance is the opposite of contra-variance. In the above example, if your parent has a parameter of type A, you can keep it as A, or redefine it to any descendant of A. This is more intuitive than contra-variance. In either scheme, a compiler can check for type-safety.

C++ and Java offer no variance for polymorphic methods. The reason for this is that if you have a routine with a different signature, even if the parameters of the parent and child are type conformant, the method overloads rather than overrides the original method. Overloading can be a major cause of confusion and errors. Many other languages require that a redefined routine must be explicitly marked as redefined or overridden.

As stated before a simple solution to the overloading problem would be to require that programmers mark the methods: `override` or `overload`. The compiler could then check for consistency, that the parameters for an overriding method are an exact, or co/contra-variant match, and that for an overloaded method, the parameters are different. Making overriding and overloading explicit is also good documentation, as it is a double check of what the original programmer really intended. Remember that overriding chooses between the alternative methods at run-time, based on the type of the owning object; overloading

chooses between the alternative methods at compile time based on the argument types.

Eiffel is an interesting case. Contrary to many strong opinions and theoretical arguments in support of contra-variance, Eiffel chooses the intuitive co-variant approach, claiming this is useful in many more situations. Eiffel has also implemented co-variance in such a way that it is type safe.

3.39 Pure Virtual Functions

Pure virtual functions provide a means of leaving a function undefined and abstract. While the concept is correct, this section shows both the syntax, and the terminology ‘pure virtual’ leave something to be desired. A class that has such an abstract function cannot be directly instantiated. A non-abstract descendant class must define the function. The C++ pure virtual syntax is:

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

This leaves the reader new to C++ to guess its meaning, even those well versed in object-oriented concepts. ‘=0’ might make sense for the compiler writer, as the implementation is to put a zero entry in the virtual table. This shows how implementation details which should not concern the programmer are visible in C++.

A better choice would have been a keyword such as ‘abstract’. Abstract should have syntactic significance as abstract functions are an important concept in object-oriented design. The C++ decision in keeping with the C philosophy of avoiding keywords is at the expense of clarity. A keyword would implement this concept more clearly. For example:

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

or

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

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

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

A function is either implemented or undefined. This to any analyst suggests a boolean state, which a single keyword conveys. A simple suggestion to fix this is to define ‘=0’ as abstract:

```
#define abstract = 0
```

then

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

Let’s look at =0 a slightly different way, as a key phrase, or a keyword which is spelt with the characters ‘=0’. If you do that, then the objection to keywords becomes a non-issue.

As for the terminology, ‘pure virtual’ is a contortion of natural language. It combines words

that are somewhat opposite in meaning. Pure means something that really is what it appears to be, as in *pure gold*. Virtual means something that appears to be what it actually is not, as in *virtual memory*. Perhaps pure virtual gold is fools gold. As has been said before, virtual is a difficult concept to grasp. When it is combined with a word such as ‘pure’, the meaning becomes more obscure.

[Stroustrup 94] gives the curious tale about the ‘curious =0’ syntax: “The curious =0 syntax was chosen over the obvious alternative of introducing a keyword pure or abstract because at the time I saw no chance of getting a new keyword accepted. Had I suggested pure, Release 2.0 would have shipped without abstract classes. Rather than risking delay and incurring the certain fights over pure, I used the traditional C and C++ convention of using 0 to represent “not there.””

Mathematically, 0 does not normally represent “not there”. Usually, 0 is just another number. Using 0 to represent “not there” leads to semantic problems which lead to many interesting discussions on topics such as 3 value and 4 value logic, etc. In the C world, there are constant arguments over whether NULL is 0 or something else. In the database world, a value is needed for “not known.” If 0 is used for “not known,” then there is a problem if the value is known, but happens to be 0. The =0 syntax is an aggregation of errors. Not only are keywords such as virtual and static overloaded, but worse a number such as zero to mean things that it does not mathematically represent.

Java and Eiffel use much clearer syntax. Java simply uses:

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

In Eiffel you specify the routine as **deferred**, meaning the details of implementation are deferred to a descendant class:

r is deferred end

The ‘end’ might look like syntactic baggage, but you can specify other abstract properties of a deferred routine in the form of pre and post conditions.

Eiffel uses the best terminology, as **deferred** means the implementation is deferred. A routine that has an implementation still has an abstract form. The abstract definition of the routine is obtained by the *short* tool, which extracts the routine signature, that is name, parameters, type, and pre and post conditions from the other details. The term abstract does not necessarily mean ‘not implemented’.

3.40 Programming by Contract

A common problem programmers face is that implementation hiding is very nice in theory, but often, you actually have to look at the internals of a class and its routines to determine what the class does and how to use it. Often you must examine the

internals of a routine before you call the routine so that it works correctly, and to determine its exact effect after the routine has executed. The signature specification of a routine is not enough; routines often have side effects.

Eiffel extends the concept of routine signature: what you must set up prior to calling a routine is documented as preconditions in the **requires** clause, and the exact effect of a routine is documented as postconditions in the **ensures** clause. The *short* tool, extracts the preconditions and postconditions with the abstract part of a routine signature, as documentation for clients of a class. Preconditions document the obligations of the caller and benefits to the called routine, and postconditions document the obligations of the called routine and benefits to the caller: hence the term programming by contract.

Programming by contract is a major technique in saving programmers from having to look at implementation code, and is most important to library vendors who don't want to give away the internals of their implementation, but do want people to buy and use their library.

Programming by contract is not just a fancy documentation scheme, but the preconditions and postconditions provide run time checks to ensure that all units of the program are behaving correctly, and thus fulfilling their contracts. This is the mechanism that detects the run-time inconsistencies discussed in the section on correctness. In Eiffel, this mechanism is integrated with the exception handling mechanism. In C++ and Java you can use assertions for run time checks, but these are not integrated into the programmers mindset as in Eiffel.

Programming by contract is the equivalent to integrated circuit specifications in the electronic component world, and also tolerances in more physical engineering disciplines. In Eiffel, the combination of static type checking with preconditions and postconditions, integrated with exception handling form a significant way to test that the software jig-saw puzzle fits together, and that the resulting picture makes sense. These techniques significantly reduce dependence on 'after the fact' manual testing.

Neither Java nor C++ have this mechanism. Another interesting case is CORBA IDL, which being an interface language for distributed objects, contract information is important. It is a glaring omission from CORBA IDL which has glaring inclusions of struct, typedef, union, etc., all of which aren't helpful in a distributed object environment, where the concept of programming by contract is even more important in considering how to connect all the system components together, and you want more confidence that the distributed jig-saw fits together. In fact this biases CORBA to C implementations. The industry should stop and think, design things carefully and correctly, and stop designing things to look like C. So often C

constructs are inappropriate, and make adopting more advanced and necessary concepts difficult.

3.41 C++ and the software lifecycle

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

This view is incorrect; someone who writes a program straight away is actually doing all the steps in parallel. It might not be the best way to do things in many circumstances, might or might not suit the style and thinking of different people, but this works in some scenarios, and can be the methodology of choice of disciplined thinkers. While that is an extreme example, the ideal way to work probably lies between that and a strictly regimented environment that assigns different people or teams to the lifecycle phases.

Some people can hold a whole problem and solution in their head and work in a disciplined fashion until the solution is complete. Mozart is said to have composed this way, producing his last three symphonies in as many months in 1788. Beethoven toiled far more over the production of his works, taking years to complete one symphony. Both composers produced masterpieces. Mozart wrote music directly, whereas Beethoven wrote themes and ideas in his famous sketchbooks. While Beethoven and Mozart had their own methods, the production of masterpieces depends on skill, not on methodologies.

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

We should cease dependence on testing. This is not to say that systematic or even random testing by an independent test group is not important, but we should rely more on better techniques in the preceding phases. Software testing can never prove the absence of error, it can only be used to detect errors if they are there.

The same people should be responsible for all stages, so that they take responsibility for the system

as a whole, rather than passing the buck and blame which occurs when analysts, designers and implementors are different groups. This is not a popular view in traditional hierarchical management structures where organisational structure is prized over quality and programmers get promoted to designers who get promoted to analysts, and managers stay aloof from the technical process, just making sure the old structure is maintained. Or even worse, those who become analysts, designers and managers have little knowledge or experience of programming and large scale software engineering. Since the second edition of the critique, Scott Adams' Dilbert comics have become widely known as accurate comments on such organisational problems. Hierarchical management discourages people from feeling responsible for a product. This culture must radically change if we are to produce quality systems.

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

So how does C++ fit into this picture? Well it is based on C that was designed mainly as an implementation and machine-oriented language. It is an old language, that did not need to consider the integrated lifecycle approach. C++ might have some of the trappings of object-oriented concepts, but it is an uncomfortable marriage of a problem-oriented technique with a machine-oriented language. It addresses implementation, but does not address other aspects of the software lifecycle so well. Since C++ is not so well integrated with analysis and design, the transformation required to go from analysis and design to implementation is costly. There is a large semantic gap between design languages and the implementation language.

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

Eiffel is specifically designed around the clusterfall model of the project lifecycle. In this model, several subparts of a project may be in different phases at any instant. It also recognises that feedback occurs from later phases to earlier phases. Eiffel itself is quite a good specification language. Its assertions and invariants are something like you would see in a formal specification language like Z. While not as comprehensive as Z, Eiffel's specification mechanisms suffice in most cases. (Bertrand Meyer was involved in the early work on Z). Thus you can use Eiffel as a documentation language in phases as early as analysis. The problem of different notations in different phases, and error-prone translation between them is removed.

The mechanism that Eiffel includes to cease dependence on testing is the assertion mechanism, integrated with exception handling. Organisations will find it difficult to make significant progress towards the higher levels of the Software Engineering Institute Capability Maturity Model (SEI CMM) until techniques such as this in Eiffel are in widespread use.

Eiffel is also integrated with a graphical CASE tool called BON (Business Object Notation) for those who feel more comfortable with classification and component relationship diagrams. Most importantly, Eiffel and BON are based on the same underlying abstract concepts. Eiffel can be generated from BON and vice-versa. This means you can easily "reverse engineer" your text, but the major advantage is that your diagrams and your text are always synchronised. There is no costly maintenance when your program changes, and diagrams have to be updated to reflect this fact. Thus Eiffel is a step towards seamless software engineering.

3.42 CASE Tools

The previous section raises the question of CASE tools. [Madsen 93] has a good discussion on graphical notation (18.8). BETA is a language that can be used for analysis, modelling and design. To a certain extent, this comes with any language that supports classes, as these are the elements of OO analysis and design, but it is important to develop the language with analysis and design specifically in mind.

If you are using both graphics and textual notations, it is important that both are based on the same underlying abstract language: text and graphics should represent the same concepts. A major problem with SA/SD was the graphical notations and programming notations were so far apart that costly and error-prone manual translation was required between the two. Unfortunately, this has set up the precedence in peoples minds that graphical and textual notations are necessarily far apart, and are surprised to see how close these are in good object-oriented systems.

It should not be thought that graphics are high level, and text is low level; that is the nature of

abstractions, not the tools or notations. In fact it should be pointed out that text is a highly evolved form of graphics; both forms of information enter our brains through our eyes. Because of the nature of graphical notations less detail can be shown. With an integrated editor detail in text can be suppressed. In identifying classes during analysis, it really makes no difference whether you document them as a series of graphical boxes with class names in the middle, or a textual list of class names. In fact many people will find the list easier to work with and later read. At any stage the notations should be interchangeable. In some cases the graphical notation will abstract away details, which is an advantage, when you don't want to see the details. As you add details though, graphical forms become unwieldy, and text is easier to manage. Unfortunately, many sectors of the industry have become convinced that graphical forms are more formal and result in magically better designs than text equivalents.

Graphics and text are best in an integrated environment. A programmer may have a class diagram as a starting point, like GUI file icons. Selecting a class will expand the class so that the interface of the class can be seen. At a different level, internal features of the class might be seen. Eventually, a level where text is seen is reached. The major failing of most CASE tools is they do not support this level of seamless integration. For the most benefit they should flow into the programming language. So called 'visual' environments do little better than putting program text in a GUI window.

Why bother with graphics then? For the simple reason that looking at the same problem in different ways aids understanding. It is also a matter of taste. Some people will find they understand graphics better, and some text. It is a good idea to cater for personal tastes, as long as there aren't too many options, in which case everyone will end up speaking their own language, and there will be no effective communication, a tower of Babel. But this has already been the case in the industry, as design methodology notations are far apart, with the analysts/designers not wanting to read programs, and programmers not wanting to read structure charts and data flow diagrams.

A common design method with C++ is to use OMT (UML) or some equivalent methodology. However, the object models are different as the graphical and textual languages are not based on the same underlying abstract language. Thus there is a semantic gap between the text and graphics. This results in more costly and error-prone development. But then as the OMT people have said "Eiffel is arguably the best commercial OO language in terms of its technical capabilities." [RBPEL91], p327. The object model of Eiffel is certainly closer to OMT than C++.

In conclusion, if CASE tools and graphical notations are to be of use, they and the programming

language must be based on the same abstract concepts.

3.43 Reusability and Communication

Reusability is a matter of communication.

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

Communication is aided by having integrated text and graphics environments, where the concrete languages of both are based on the same underlying abstract languages, or object models. Communication is also dependent on clear and clean syntax.

As C/C++ suffer from arcane and cryptic syntax, it does not support the goal of clear communication.

Java cleans up a fair bit of C/C++. The mess that is caused by the preprocessor is removed. However, Java still suffers from some of the deficiencies of C in this regard.

Eiffel has been designed with communication in mind, and is not bound by the shackles of C syntax. It borrowed from the clean syntax of Ada. Along with the Eiffel syntax were designed style guidelines, so the Eiffel syntax lends itself to a clear style.

Eiffel also has utilities like *short*, where the abstract interface of classes can be extracted from the full details.

Eiffel provides an extra significant mechanism, that of integrated assertions. The short tool will extract the assertions with the interface descriptions. This has been described in the section on programming by contract. Programming by contract helps decide whether a class is useable in a new situation, and then how to use it, so this is an important tool for communicating the purpose, intent and correct usage of a software module. Thus assertions are very much a courtesy concern.

Reusability is well supported with clear communication in Eiffel.

3.44 Reusability and Trust

Reusability is a matter of trust.

Building trustworthy components is a safety concern. Trust results from confidence that safety concerns have been met. If you do not have confidence in a software component, then you won't want to reuse it. You could doubt that the software component provides enough functionality, or correct functionality. You could doubt that the component

is efficient enough, or worse it might fail. As so many traps in C++ result in ‘bugs’, it is difficult to trust a software module, so it is less reusable.

In the real world of reusability, the ideal of trusting programmers is inappropriate, and results in less trustworthy software; in reality, customers doubt the claims of suppliers. It is the onus of the supplier to prove their claims, and thus trustworthiness of the software. The client is not required to trust the supplier’s programmers. Potential clients of a software component, require assurance that the component is trustworthy.

Trusting programmers is against the commercial interest of both parties. This is not to cast dispersion on programmers, but merely recognises that computers are good at performing mundane tasks and checks, but people are not. If people were good at such things, we would not need computers in the first place.

Even though you might not trust your programmers, this is not an excuse to employ anything but the best skilled programmers, and programmers should also be given the best training. Consider a Stradivarius violin: it will sound bad in the hands of a bad violinist. But a good violinist will insist on a Stradivarius, rather than a cheap brand where he won’t sound his best. In computing, we frequently argue whether it is the tools or the programmers. It is a combination of the two; if either is lacking, trustworthy software will not result.

Java “eliminates entire classes of programming errors that bedevil C and C++ programmers” [Sun 95]. This means that you can better rely on externally developed Java packages.

Eiffel also is not bedevilled by the same classes of errors. Thus you are more likely to produce software that can be used in other contexts, and be able to find software that can be reused in your context.

Eiffel assertions are also important here. As assertions are checked at run time, they ensure that the software is working correctly, so the level of trust in external components is higher, and you reuse them with more confidence.

3.45 Reusability and Compatibility

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

This can be solved in two ways: firstly, a library vendor could provide the entire source of a library so it can be compiled with the customers compiler; if the sources are proprietary the vendor will need a separate release for every environment, and every compiler in that environment.

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

An important feature of Java is that it is architecture neutral as Java compilers produce byte code instructions for a virtual machine. Java provides a “universal intermediate machine readable representation of programs” as I called for in this paper’s second edition.

Eiffel implementations provide a high level of source code compatibility. However, the generated C from different implementations can have different object layouts. Thus a class library will have to be recompiled if it is to be used in a system compiled with a different vendors implementation.

Another form of incompatibility between libraries is incompatibility of type definitions. A glaring example in C++ is the number of ways the simple type *boolean* can be defined. For more on this see the section on booleans.

3.46 Reusability and Portability

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

Java is also the winner in this category, due to its virtual machine, and removal of pointers. Eiffel code is also highly portable, but you are currently confined to systems where Eiffel compilers exist of which there are many. As most Eiffel compilers generate C, you can port the generated C to platforms where there is no Eiffel compiler. With Java, only a virtual machine interpreter needs to be available on the system in order to run Java programs.

As the Java virtual machine seems to be sufficiently semantically rich, it could be that other languages target the Java virtual machine, and that it becomes a universal machine code. Such a marriage might not be as easy as it appears, if the object models of different languages are sufficiently different from the Java model. Sun does seem to have kept the virtual machine independent of physical object layout, and any assumptions that would make this too hard.

3.47 Idiomatic Programming

The ability to program in different idioms is argued as a strength of C++. Idiomatic programming, however, is a weak form of paradigmatic programming; it is programming in a paradigm without necessarily having compiler support for that paradigm. The compiler cannot check for inconsistencies with the idiom, or paradigm. Defines can often be used to invent idioms. Anyone who has attempted to do object-oriented programming in a conventional language using defines will realise that it is impossible to realise the benefits easily, if at all, without compiler support.

Both Java and Eiffel are strongly object-oriented: the idiom is OO. You don't have to bring together various sub-projects each of which might have used their own favourite idiom.

3.48 Concurrent Programming

The object of concurrent programming is that computing resources can be harnessed to efficiently compute problems that would otherwise be inefficient to compute using a single processor. In the next ten years multiple processor arrays that execute programs concurrently will likely become common. Concurrency requires much cleaner languages, than the single processor languages of today.

Object-oriented concepts support concurrent programming. Objects can execute state changing code independently of each other. Concurrent programming will be enabled by the division of the state space of a system into modules to achieve a high degree of independent processing. Objects provide a scheme to cleanly divide state spaces. The demand that everything be divided into loosely coupled modules, that only interact through well defined interfaces might be perceived as inefficient; but it is precisely this scheme that will mean that concurrent solutions can be developed efficiently and transparently to the programmer.

Concurrency should be transparent to the programmer, as concurrency is a low level implementation consideration; concurrency is how a computation is done, not what is to be computed. However, there are examples where concurrency is manifest in the problem domain, such as many simulation problems like multiple queues, for example check-outs in a supermarket. The implementation issue of concurrency is how processes are allocated to processors. The programmer should not be concerned with this, rather what is to be computed, not how. How something is computed is the concern of the target environment, ie., the compilers, operating system, and hardware.

The aim of concurrent processing is to keep all the processors in a processor array as fully utilised as possible, so that processor resources are not wasted. There is nothing more mysterious to concurrent programming than the efficient use of

resources. Keeping all processors busy is an inherently dynamic problem, which the programmer cannot determine statically at compile time. All the processors can be kept busy, as long as there are enough threads in the system.

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

Object level is natural for the programmer, and has the advantage that a programmer can implement a system without taking into account parallel processing at all. The same program will run and produce identical results irrespective of whether the customer is running a single processor, or a processor array. This way the programmer concentrates on the model and design of the problem, not on deployment concerns.

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

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

It might not be impossible to implement concurrent processing in C++, but it is difficult as in many ways C++ is not suited to concurrent processing.

Java provides threads. It also removes C features like globals that are problematic to concurrency.

Eiffel has a recommendation [Meyer 96c] that extends Eiffel with a single keyword **separate** to provide concurrency. Both Java and Eiffel have simple concurrency mechanisms due to their cleaner base than C++.

3.49 Standardisation, Stability and Maturity

Object-orientation is now nearly 30 years old, since Simula 67. Smalltalk is about 20 years old, Ada 95 is only one year old, but based on Ada 83, which is about 13 years old. C++ is 13 years old. Eiffel is 10 years old, and Java is just one year old.

The age of a language does not relate to its stability and maturity. Java is the youngest language, but Java appears to have a well thought out and stable language base, also having a comprehensive set of OO libraries. Thus Java is off to a good start, but only time will tell. It already has quite a number of books.

Ada 95 is one year old. But that is one year since the standard was ratified, so it is a good deal older than a year. Ada 95 is the product of an ISO/ANSI/DoD standard. Thus Ada 95 vendors have a very stable base from which to implement. This gives Ada 95 a good start over other languages, where there might be implementations, but they are shooting at a moving target.

Eiffel is not subject to the 'formal' ISO/ANSI standards; it has its own non-aligned standards body NICE (Non-profit International Consortium for Eiffel). Eiffel is now in its third incarnation, Eiffel 3 that is fully described in *Eiffel: The Language* [Meyer 92], the Eiffel equivalent of the C++ ARM. However, the definition of Eiffel 3 has been very stable since 1992, requiring only a few extra validity rules, and small clarifications: Eiffel is probably the best designed language ever intended for commercial use. The largest change to the language is now under consideration, which is to add the **separate** keyword to allow support for concurrent and distributed processing. This will not affect existing programs, and early releases of implementations with this mechanism are now available. Eiffel also has a standard library. The standard library is more changeable than the base language, but is also under the control of NICE. Thus Eiffel has attained a great deal of maturity over 10 years, and the standards are very stable. This gives Eiffel a considerable advantage in that libraries are much easier to update to address new and changed requirements than compilers. Therefore, Eiffel should evolve more quickly into new problem domains, without the traditional resistance from compiler vendors.

The most serious problem that Eiffel has faced in the past was stability of implementations. As Eiffel is an ambitious language and environment, many new and difficult concepts have been pioneered and made into industrial strength packages. Eiffel is very demanding on compilers, which need to do things like global analysis, which is an issue that C++ conveniently avoids. Eiffel does not concede to compromises which place burdens on the programmer in the same way that C++ does.

However, stable forms of Eiffel environments are now becoming widely available. In 1996 Tower Technology has released version 2 of its compiler

and environment, and ISE has announced version 4 of its environment, which addresses many issues that users did not like previously, and now includes menus and other facilities, which gives it a more Macintosh/Windows look and feel. SIG Computer has also announced its *Visual Eiffel* for release October 1996. There is also an independent experimental version known as SmallEiffel, which can be downloaded for free.

Another problem that Eiffel has had is the lack of titles. [Meyer 88] is the classic book on OO, however, it is based on Eiffel 2.0, not version 3. Meyer's next book "Eiffel: The Language" [Meyer 92] is the language lawyer's reference, but it is possible to navigate for an overview. However, there are now over ten titles on programming in Eiffel, quite a few of which are used to teach university courses on OO.

Smalltalk is now a widely used language, and has proven to be very effective in some environments. Different implementations of Smalltalk do not share libraries, and do not interoperate.

Out of all the languages here, C++ although 12 years old, provides the fastest moving target for vendors. It is claimed to be standardised, as it is subject to ANSI/ISO standardisation, but this work is still very much in progress. You can check status of the standard on the X3J16 WEB page in the WEBliography). The number of issues to be addressed by the committee keeps increasing, rather than decreasing. C++ was submitted to the standardisation process too early, and the committee has had to do too much design work that should have been done before C++ was submitted to the standardisation process.

The committee hopes to progress the standard to CD (Committee Draft) this year (1996). The FAQ shows a timetable which will produce an IS by December 1998 (see WEBliography: http://reality.sgi.com/employees/austern_mti/std-c++/faq.html#B8). After IS is achieved, it will probably be several more years before a significant number of vendors are fully compliant. By that stage, users will probably be clamouring for more features and fixes to old problems. I have already heard stories of C++ tool vendors complaining that the standard is too horrendous to understand, and then to implement anything compliant. Standardisation should stabilise the specification, but C++ has continued to become less stable. The fact that the C++ standard is so unstable indicates that the C++ committee realises there are many shortcomings in C++ that they must rectify. There are many flaws that the committee knows about that I do not cover in this critique, but also many of the flaws that are covered in this critique, the committee have no intention of addressing, as that would break too many existing programs and C compatibility.

In the preface to [Stroustrup 94], Bjarne Stroustrup writes "C++ is still a young language. Some of the issues discussed here are yet unknown

to many users. Many implications of decisions described here will not become obvious for years to come.”

Coming to consensus in the C++ world is a difficult task. [Stroustrup 94] states this frustration as “Dealing with stubborn old-time C users, would-be C experts, and genuine C/C++ compatibility issues has been one of the most difficult and frustrating aspects of developing C++. It still is.”

Many comments in [Stroustrup 94] show that C++ is still a moving target. Garbage collection is mentioned as “when (not if)”. Thus when GC is fitted to C++, developers will be faced with quite a transition in paradigm. All of this uncertainty in C++ might keep the programmers busy, after all many of them want to code exclusively in C++, while ignoring all else; but it will be very costly for the companies that are locked into C++.

There are still unresolved things the X3J16 committee must sort out, especially in the area of C compatibility. [Stroustrup 94] says “The “compatibility wars” now seem petty and boring, but some of the underlying issues are still unresolved, and we are still struggling with them in the ANSI/ISO standards committee. I strongly suspect that the reason the compatibility wars were drawn out and curiously inconclusive was that we never quite faced the deeper issues related to the differing goals of C and C++ and saw compatibility as a set of separate issues to be resolved individually.” Since C compatibility results in so many problems, serious consideration should be given to this basic tenet of C++.

The C++ community seems to think using a fundamentally flawed tool is acceptable and that the rest of the world must wait for them to straighten these issues out, which in many cases isn’t even possible. It is also a hidden cost to companies that their programmers must continually keep up to date, and abreast of the arguments for and against certain constructs. Many other languages have solved these problems.

As a postscript to this section, I will remark that a lot of argument for or against particular languages seems to come from people who believe that there will be an eventual winner in the evolution of languages, and they want it to be their favourite, so will fight for dominance. I can see no evidence that this will happen. I think new languages will continue to be invented: some will be based on continuing mistakes from old languages while adding new features for compatibility; others will avoid previous errors while adopting new paradigms. I can’t see that the programming language world will ever become stable. If people in the industry can accept that, then we will have programmers that are more amenable to change language, being able to use the language that is best suited for the purpose, and the maturity of language criticism will improve, as we see each language as a passing phase, to which we owe no long term allegiance.

3.50 Complexity

There are several kinds of complexity. This critique focuses mainly on the complexity of the C++ language itself. When considering complexity, one needs to consider the complexity of the development task as a whole. The complexity of the language might only be a small part of that.

Apart from the language, we need to consider the programming environment, that is editors, tools for example make, etc., the methodologies and tools, and the supporting libraries.

With C++ the conventional wisdom is often to use a methodology such as OMT. Here the concepts of the methodology do not exactly match the concepts in the programming language. Thus you have a semantic gap, where translation must occur. This translation is costly, and frequently ends in specifications that do not match what was eventually implemented.

Both Eiffel and BETA see it as important to develop their methodologies and graphical notations based on the same underlying concepts. The importance of this integrated approach should not be under-appreciated.

As for environments, [Stroustrup 94] has the following to say: “Every language in nontrivial use grows to meet the needs of its user community. This invariably implies an increase of complexity. C++ is part of a trend towards greater language complexity to deal with the even greater complexity of the programming tasks attempted. If the complexity doesn’t appear in the language itself, it appears in libraries and tools. Examples of languages/systems that have grown enormously compared to their simpler origins are Ada, Eiffel, Lisp (CLOS), and Smalltalk. Because of C++’s emphasis on static type checking, much of the increase in complexity has appeared in the form of language extensions.”

“C++ was designed for serious programmers and grew to serve them in the increasing large and complex tasks they face.”

P.J. Plauger in [Plauger 93] argues that the complexity of C++ has put it on par with PL/I, Ada (83) and Algol 68. He does not accept the complexity in C++ as a good thing. Criticising the complexity of Ada is somewhat unfair. An amount of Ada’s complexity is due to its support of multitasking and real-time programming. Simula also has facilities for co-routines and processes, and Ada and Simula are reasonably unique for their inbuilt support of these facilities. In the 1980s, the need for such facilities was not widely recognised. However, the need for concurrency and distribution is now becoming recognised.

Another feature of Ada that might contribute to the perception of complexity is genericity. Again the charge that this makes the language over complex is based on not understanding genericity. I have already covered this topic in the section on templates. Thus Ada has been criticised for being complex, but most of this criticism is due to not

understanding essential features such as genericity and concurrency.

Many C programmers have been guilty of dismissing features they don't understand as complexity, and Ada has been a favourite target. I am not saying that Plauger is in this category, as he makes some valid points about Ada. But the accusation of complexity against Ada should not be overstated as it has too frequently emotionally been in the past. In the computing industry, there is a low level of understanding and experience that one must have before becoming an expert or vocal critic, particularly of languages like Pascal and Ada.

C++'s complexity is not solely due to static type checking. Eiffel is more strongly type checked than C++, but doesn't suffer from the same complexity problems.

As for the environment. The burden of environment is far less for the cases of Eiffel, Java and Ada 95. In Eiffel, a separate simple language exists, LACE to specify to the compiler how to compile the program. This contains such things as environment variables, debug and other options, etc. It also provides the basis for *separation of concerns* so that environmental details are completely removed from the Eiffel language. Eiffel is also integrated with complete editing and development environments.

Java has removed such environmental considerations as `#include` and `make`. Edmond Schonberg writes that the environmental baggage for Ada and Ada 95 is far smaller than C++ (see WEBliography for his Ada contrast to C++).

The Eiffel libraries are very large and comprehensive; but this only reflects the richness of data structures that exist, and the number of application domains. Eiffel libraries are available for networking, compiling and parsing, Windows programming as well as platform independent user interfaces and many other things. The Eiffel libraries simplify naming complexity by standardising the vocabulary between classes. For example, `put` is used to enter an item in any collection data structure like `ARRAY`, `LIST`, `QUEUE`, and even `STACK` where the routine would normally be named `push`. The libraries enable the complexity of specific domains to be removed from the language, which is simple and yet general purpose.

Smalltalk also has a large library, which extends an otherwise small and simple language. Classes that a programmer adds also become part of the Smalltalk environment.

Java also provides a comprehensive library to deal with many aspects, including `java.net`, `java.awt` (abstract windows toolkit), etc. Eiffel, Smalltalk and Java do not ignore the issue of complexity; they put it where it should be: in the libraries. In terms of complexity, they implement Stroustrup's principle that "what you don't use, you don't pay for." In C++ you pay very much for complexity, as it is in the language.

C++ can to some extent be extracted from the complexity of its environment. But as long as the mechanisms of `#include` persist, the environments that C++ is ported to will have to adapt to the C/Unix way of doing things. Where the environment is separate from the language, there is no environmental adaptation that needs to be done, and less retraining of programmers for each environment they need to program in.

I can accept that C++ was designed for serious programmers. However, Ada 95 and Eiffel are both designed for the serious software engineer. (Java remains to prove itself in this arena.) Eiffel in particular shows that complexity can be dealt with in a serious industrial strength software engineering environment.

Complexity is not the necessary companion of seriousness. This does not ignore the complexity of any application domain; in fact it enables you to focus on the complexity of the programming task in hand, not on the complexity of the tool.

3.51 C++: the Overwhelming OOL of Choice?

This headline comes from Cutter Information Corps "Object-oriented Strategies" May 1996 edition. Based on their findings, C++ accounts for 80% of all OOLs, with Smalltalk running a distant second at 11%. They claim that in 1995 OO software development products hit \$1.3 billion. However, let's examine how C++ is used: many C programmers have not wanted to touch C++, but they do use a C++ compiler to compile their C. This greatly exaggerates the market penetration of C++ and the size of the OO market, so it is impossible to determine the true market penetration of OO. You are not doing OO just because you are compiling with C++.

Microsoft and Borland have put most of their development environment energies into C++, so this makes it attractive to buy a C++ environment, even if you are just programming C. Probably the true number of C++ installations being used for OO would be between 10-50%, which cuts down the size of the OO market by a large amount, the size of C++'s predominance in that market, and means the other OOLs in the market have a much higher significance than Cutter makes out. Smalltalk and Eiffel are pure OOLs, so every one of their sales you can count as an OO installation, whereas the same is not true of C++. Measured C++ sales are riding on C's success. C++'s success is less than overwhelming. It is a marketing success, rather than a technical or programming success. Companies using C++ are paying for it with longer cycle development times, and less reliable end product.

One way a manager might perceive C++ to be a winner is the sheer number of books one sees in a bookshop on C++. This is matched by a huge number of courses. An observation about the nature of many of these books is that they are often titled something like "How to build a widget in C++," or

“Compiler Construction in C++.” “Books appear like mushrooms after rain” [Plauger 93].

The mushrooming book market is a great boon for publishers, as it implies that for every possible software artefact you can build, they can publish a book about it in every possible programming language. All you really need is the books “Programming in C++,” and “How to build widgets,” or “Compiler Principles and Construction.” Then your programmer needs experience, lots of it. Don’t be fooled by this trick to get a high title count.

Many C++ books are on how to avoid the traps and pitfalls, and develop rigorous coding standards, which might appeal to management as the solution, but they don’t solve the root cause of the problem. Making sure everyone is well trained and versed in these style standards is an expensive and usually ineffective band-aid measure, especially where different companies have different standards and expectations, so you need to retrain every new recruit, who will probably decide they don’t like your way of doing it anyway, and leave after a short period. Of course you can satisfy yourself that his dissatisfaction was due to his inappropriateness for your organisation, which is better organised than most. After all, you are ISO 9000 accredited and are turning out a very successful line of ‘concrete life-jackets’ (a Tom Peters quote).

[Sakkinen 92] observes the “Endemic C++ Culture.” He notes that too many courses on “design” have the appended clause “with C++.” This is because C++ has its own curious terminology, which is in many ways different to the rest of the OO world. He makes a case that concepts and principles should be taught, then how to map them onto any particular language.

Of course books are aimed at different audiences: professionals versus those who just program for a hobby; those who have an academic interest in languages; implementors of compilers and other language processing tools, who need formal non-ambiguous statements about how the language works; beginners versus those for whom this is their fourth or fifth language. C++ should not be for beginners, as it is better to learn the principles from a clearer language than be confused by what all the syntactic *knobs and dials*, and superfluous constructs do in C++.

As for courses, C++ has proven so difficult to learn that you need lots of courses. Not only do you need to learn the language, but the complexities of the environment add an even more substantial overhead. It will probably be best to start on C++ with a course. However, with simpler languages such as Java and Eiffel, buying a good book, and self experimentation will quickly cover every aspect of the language. It is a bonus if you can get a course, but it is not essential to get started.

4. Generic C Criticisms

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

4.1 Pointers

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

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

Pointer arithmetic is error prone. Pointers can be incremented past the end of the entities they reference, with subsequent updates possibly corrupting other entities, which is a major source of the undetected inconsistencies, which result in obscure failures, discussed in the section on correctness. In the STL library, iterators are provided as the generalisation of C pointers for access to elements of structures such as arrays.

Programmers can by-pass encapsulation with pointers; C undermines OOP by providing a mechanism where state outside an object’s boundaries can be changed. Since pointers are intrinsic to writing software in C this exacerbates the problem. Pointers as implemented in C make the introduction of advanced concepts like garbage collection and concurrency difficult.

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

relocated, only the master pointer needs to be updated.

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

Simpler environments might not provide object relocation, so double indirection would be an unnecessary overhead. In order for programs to be portable and efficient in different target environments, such system details should be the concern of the target compilation system, not of the programmer.

C’s pointer declaration syntax causes another small problem:

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

This does not mean, as might be easily read -

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

but

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

and should be written thus to avoid confusion.

Java has abolished pointers as “Most studies agree that *pointers* are one of the primary features that enable programmers to put bugs into their code. Given that structures are gone, and arrays and strings are objects, the need for pointers to these constructs goes away,” [Sun 95]

Eiffel also has no pointers only object references. In Eiffel, the exact referencing mechanism does not matter. For example in the expression *x.f* the reference *x* might be a pointer to an object in the same address space, or it might be an Internet address of an object. References enable the location and access method of an object to be transparent.

4.2 Arrays

Page 137 of the C++ ARM notes that C arrays are low level, yet not very general, and unsafe. Page 212 admits, “the C array concept is weak and beyond repair.” Modern software production is less dependent on arrays, especially in the object-oriented environment. The trade off to be optimal, rather than general and safe no longer applies for most applications. C arrays provide no run-time bounds checking, not even in test versions of

software. This compromises safety and undermines the semantics of an array declaration, i.e., an array is a particular size, and can only be indexed by values within the bounds of the array. The array size might not be determined at compile-time, but dynamically at run-time. An index to an array is a parameter in the domain of the array function. An index out of bounds is not a member of the domain, and should be treated as severely as divide by zero. But in C this is another significant source of undetected inconsistency, which can result in obscure failures.

C has no notion of dynamically allocated arrays, whose bounds are determined at run time, as in ALGOL 60. This limits the flexibility of arrays. You cannot resize C arrays. Multidimensional arrays are only really one dimensional. You cannot individually resize the rows of a multidimensional array. The C definition of arrays compromises both safety and flexibility.

There are many ways you can undermine arrays in C and C++, as an array declaration is really just equivalent to a pointer. The following example comes from [GWS 94]:

```
char *str = "bugy";
```

then the following are true:

```
0[str]      == 'b';
*(str+1)    == 'u';
*(2+str)    == 'g';
str[3]      == 'y';
```

This is amazingly flexible syntax for something as inflexible as C arrays, which is against Meyer’s “Principle of Uniqueness” (see introduction), providing several ways to do the same thing, but still not doing it particularly well.

The unsafeness of C arrays is shown in the next example:

```
#include <stdio.h>
#include <string.h>
main ()
{
    char str[] = "TEST";
    char *p = "TEST2";
    const char str3[] = "TEST3";
    char *p3;

    printf ("str = %s p = %s str3 =
           %s\n", str, p, str3);
    p3 = &str;
    strcpy (p3, "some junk");
    printf ("str3 = %s\n", str3);
    str[6] = 'X';

    printf ("str = %s p = %s str3 =
           %s\n", str, p, str3);
}
```

The results (at least from my C compiler) are:

```
str = TEST p = TEST2 str3 = TEST3
str3 = junk
str = some Xunk p = TEST2 str3 = Xunk
```

One view of arrays is just another object-oriented entity which should be treated in an object-oriented manner as a class of data structure. It should have interface definitions, and consistency checks inherent in object-oriented systems. Another view is that an array is an implementation of a function, where pairs of values explicitly map the domain uniquely to the range, rather than being computed. This suggests that Algol was incorrect in syntactically distinguishing arrays by using square brackets. An array just maps the input argument (the index) to the value stored in that location in the array.

[Ince 92] considers that arrays and pointers need not be relied upon so heavily in modern software production, as higher level abstractions such as sets, sequences, etc., are better suited to the problem domain. Arrays, and pointers can be provided in an object-oriented framework, and used as low level implementation techniques for the higher level data abstractions. Ince suggests that arrays and pointers should be regarded in the same way as *gotos* in the seventies. He suggests that languages such as Pascal and Modula-2 should be regarded in the same way as assembler languages in the seventies. This applies even more to C and C++, because pointers and arrays are far more intrinsic in the use of C and C++, with lower level, less flexible arrays. Although Pascal arrays are weak compared to those of ALGOL, they are still much better than C arrays.

In both Eiffel and Java, arrays are first class objects. Both languages have no need of the `sizeof` function. In Java to get the size of an array you use `myArray.length`. In Eiffel this is `my_array.count`. Arrays can also be resized.

Both Eiffel and Java provide bounds checking on arrays. Java's checking is built-in. Eiffel's checking is integrated with the assertion mechanism.

Eiffel goes a step further in array element access. You access an element with the *item* function as follows:

```
v := my_array.item (i)
```

This can also be accessed by an infix operator, `@`:

```
v := my_array @ i
```

The *item* function is defined as:

```
item (i: INTEGER): G
  require
    lower <= i;
    i <= upper
```

This shows how Eiffel's assertion mechanism is used to document semantics in the interface, as well as for a checking mechanism.

4.3 Function Arguments

Arguments are a fundamental mechanism for reuse in software construction. Without arguments you would be forced to write a different routine for every possible input parameter. Arguments allow one algorithm to be reused on sets of input values.

Arguments pass routines simple values (by-value arguments), or references to entities (by-reference arguments). (Actually, there are more possibilities than this. [Hext 90] is an excellent text on the possibilities.) Arguments are inputs to routines, and should not be changed. When memory was expensive, reusing parameter space could conserve space. Changing arguments, however, is semantic nonsense, and most languages get this wrong.

By reference arguments enable a routine to change the value of an entity external to the routine. Such updates beyond the environment of a routine are side-effects. This introduces a mechanism of updating the state space, other than straight assignment (although the routine can use assignment to achieve the 'dirty deed'.) The problem is that the state of an object can be changed without using the well defined interface of the object, so encapsulation is compromised. By-reference arguments should not be used to change external entities. Values should only be passed to external entities by the return value of a function. Semantically, this is different to assignment to a reference parameter; data flows through the program in one direction, in via arguments, and out via return values. Mathematically this maps a value of an input type to a value of an output type. Both input and output types can be compositions of other types, ie., $f: I1 \times I2 \times \dots \times I_m \rightarrow O1 \times O2 \times \dots \times O_n$. Abstract data types can be used to design such systems. This will also help target environments to increase parallelism and concurrency in a way transparent to programmers.

In object-oriented programming, by reference arguments are used to pass the original object, not a copy. The called routine, however, should not change the state of the referenced object. Only calling a routine in the passed objects interface can change the state, although introducing side effects into arguments like this is dubious and should be avoided. Passing objects by-reference has the desired effect of the object being given to you, without being yours to change, although you can effect change in the object. C++ does have a nice concept called *const correctness*, which provides a modifier on arguments `const` which disallows any changes to that argument.

C shares faulty arguments with many other languages. The interaction of C's pointer mechanism with a faulty parameter mechanism, however, makes C considerably worse than most other languages. In C, pointers are used to simulate

by-reference arguments with by-value arguments. The programmer must perform tedious bookkeeping by specifying `*s` and `&s` for referencing and dereferencing. Distinguishing between by-value and by-reference arguments is not just a syntactic nicety, included in most high level languages, but a valuable compiler technique, as the compiler can automatically generate the referencing and dereferencing, without burdening the programmer. Again C adopts operators to provide the functionality, rather than a declarative approach, which would centralise decisions and let the compiler do the rest.

In Java arguments can only be passed by-value (as in C). However, there are no pointers, so passing by-reference cannot be simulated.

Eiffel routine arguments are read-only. This means that they are pass-by-constant which is stronger than pass-by-value, where the arguments are treated as local variables which may be updated, pass-by-constant disallows this.

4.4 void and void *

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

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

When an entity is assigned to a reference of `void*`, it loses its type information. When it is assigned back to a typed reference the programmer must explicitly specify the type information with a type cast. This is error prone and should at least result in a run-time check. Without a runtime type check, the routines of one class can be mistakenly applied to objects of another class, which results in undetected inconsistencies leading to obscure failures.

As [Stroustrup 94] points out: “having `void*` unsafe can be considered acceptable because everybody knows - or at least ought to know - that casts from `void*` are inherently tricky.”

Interestingly, `void*` is the exact opposite of void, so yes this is a programming oxymoron. Void means no object of any type; that is the empty set. `Void*` on the other hand means any object of any type; that is all objects of the all encompassing set, or the universal set of all objects that can exist in a system. So void and `void*` represent complementary sets.

Eiffel and Java both provide a class that is at the root of the inheritance tree. In Java it is `Object`,

and in Eiffel it is *ANY*. Any object can be assigned to a reference of these types. In C++ this is provided by `void*`, but `void*` is not at the root of the inheritance tree, hence its type unsafeness.

Eiffel also defines the type *NONE* at the bottom of the inheritance tree, which is a class to which no objects belong. *NONE* is the complement of *ANY* and vice versa. Type *NONE* has the single value *Void*, which signifies no object. *Void* is the equivalent of 0 (meaning NULL) in C++. This means that Eiffel’s type system is more consistent, as *ANY* and *NONE* reside within the type hierarchy at the top and bottom respectively. However, `void` and `void*` do not fit into the type hierarchy in C++.

4.5 void fn ()

The default return type of a function is `int`. A typeless routine returning nothing should be the default, but this must be specified by `void`. Syntactically no `<type>` suggests nothing to return. This is an example of where C’s syntax is not well matched to the concepts and semantics. Also a typed function can be invoked independently of an expression, which is a shorthand way of discarding the returned value, but compromises type safety. Using a typed function as a void should result in a type error.

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

[Stroustrup 94] also voices the opinion that default `int` is bad. He had tried to make the type specifier explicit, but was forced to withdraw by users: “I backed out the change. I don’t think I had a choice. Allowing that implicit `int` is the source of many of the annoying problems with C++ grammar today. Note the pressure came from users, not management or arm-chair language experts. Finally, ten years later, the C++ ANSI/ISO standard committee has decided to deprecate implicit `int`.”

One improvement in Java is that the result type of the method is not optional. That is you don’t get `int` by default. Otherwise, Java does not clean up most of the deficiencies of C. In order to specify a procedure rather than function, Java still requires the `void` specifier. Java does discard the C term function (which was wrongly used anyway), but

makes the situation no better by calling both procedures and functions *methods*. Thus there is no clear distinction between procedure and function. Java also allows you to ignore returned values.

Eiffel uses the term *routine* for called units of code and distinguishes that there are two kinds of routine, *procedures* and *functions*. It is recommended practice that only procedures change object state, and functions do not. Functions always return a value. That is they follow the mathematical definition of function that takes a value of one type (the type may be compound, hence multiple arguments), and maps it to a value of another type.

4.6 fn ()

We have already seen that C functions are a poor cousin of mathematical functions in the section on inlines. C functions expose implementation detail; that is, whether an entity is implemented as a constant, variable or value returning routine. C functions are different to the mathematical concept of a function. C functions are really parameterised invokable code, which other languages call procedures, subroutines, etc. Java calls them *methods*. Data can be accessed functionally in the mathematical sense, but this is different to insisting that all data is accessed through a C function. Functional access to data really means that data can only be retrieved, not assigned to.

Empty parentheses represent the function call operator in C. Even though '()' is mathematical looking, it is semantically equivalent to FORTRAN's CALL, COBOL's PERFORM, and JSR in assembler. The design of these operators was influenced by the underlying machine architectures. The function call operator is low level, machine and execution oriented, and in the 'how' domain. True high level languages require no such operator, the compiler realises from the declaration that the entity referenced is a function and automatically generates the machine call operator.

This is opposite to most Unix shells, where invocation operators such as 'run' and 'exec' are not needed. One of the nice things about Unix shells is that the set of in-built commands is extensible. The ability to execute file names as commands extends the command repertoire. The shell runs executables and interprets shell scripts. Unix shells do not distinguish between inbuilt commands, shell scripts and executable programs. This is a widely accepted as an elegant and effective convenience. C's () operator introduces the equivalent of a run command into the language.

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

Languages that have an invocation command or operator have an unnecessary distinction between value returning routines and constants and variables.

It is trivial for a compiler to provide transparency of view for constant and variable access and function invocation. In ALGOL style languages, the compiler automatically deduces invocation when it sees a name that was declared as a routine, rather than a variable. The compiler knows that the identifier refers to a routine because the compiler stores much information about an entity. A compiler can check that the programmer uses the entity consistently with the declaration. A compiler can generate correct code, without burdening the programmer with having to use an explicit invocation operator. This enhances flexibility and implementation independence.

Variables and functions should be interchangeable for optimisation. '()' is a good example of where the operator approach of low level languages adversely affects flexibility as opposed to the declarative approach of high level languages. In C, it is not possible to change a function to a variable without removing all the '()', or a variable to a function without adding '()' to all the invocations. This might be spread over many files, and the programmer might not bother with optimisation to avoid the tedium of the task. So the () operator reduces flexibility. The () operator is another bookkeeping task imposed on the C programmer. The C++ recommended style is to code superfluous accessor functions to blur the distinction. Pure functional languages such as SML remove the variable/function distinction altogether, by not having variables at all.

Java has made no improvement here. The visible implementation difference between variables and functions remains. Eiffel removes this distinction as constants and variables are accessed functionally. A programmer can flexibly change a variable to a function in a class interface and vice versa for optimisation or extension, without the need for all clients to change their code. Thus even though changes have been made, the class interface remains unchanged.

C also has pointers to functions. Function pointers are analogous to the call by name facility in ALGOL, and this was recognised as having pitfalls. Consistent application of the object-oriented paradigm avoids the need for function pointers. A common use of function pointers is to explicitly set up jump tables. Jump tables are the mechanism behind virtual functions. The design of a program can take advantage of this fact, without resorting to explicit jump tables. Another use is to jump to a function in a table that is indexed by an input character. A switch statement can cater for this mechanism that makes what is meant explicit, while keeping underlying mechanisms (and possibly optimisations) transparent. C++ allows function pointers to member functions to be stored in tables (via the .* and ->* operators).

4.7 fn (void)

In C $f()$ means the function f can take any number of arguments of any type without type check. ANSI C has adopted $f(\text{void})$ to mean a function that really has no arguments. C++ sensibly differs from this in that $f()$ now means a function that has no arguments [Stroustrup 94].

4.8 Metadata in Strings

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

In C strings, metadata about where strings terminate is stored in the string data as a terminating null byte. This means that the distinction between data and metadata is lost. The value chosen as the terminator cannot occur in the data itself. Since inserting a null is often the responsibility of the programmer, not the run-time environment, there is the potential for more undetected inconsistencies resulting in obscure failures.

A common alternative is to store a length byte in a fixed location preceding the string as Pascal does. The advantage is that the length of a string is easily obtained, without having to count the number of elements up to the terminating null. Another advantage is that 0 is a valid value in a string. This implementation is hidden from the programmer and other methods could be used without the programmers having to change the program. C's null terminator makes the implementation visible to the programmer.

Java's strings are first class objects. You can't determine the length of a string by scanning for a null. You use the *string.length* method (function). Eiffel's strings are also first class objects.

4.9 ++, --

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

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

For full generality, only the first form is required; the last two forms *a++* and *++a* are the postfix and prefix forms, which can be used in the context of another expression. Thus several updates can be performed in one expression. This is a very powerful and convenient feature, but introduces side effects into an expression that sometimes have

surprising effects, and can lead to program errors. The following example is given on p.46 of the C++ ARM -

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

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

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

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

In fact constructs such as:

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

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

Memory update is a problematic, but necessary part of programming. A language should provide it in a consistent and expected way. Many languages recognise that memory update is problematic, and

typically only provide the assignment operator as a sufficient update mechanism. (Many languages have block memory copies as well, but assignment can provide block copy.) Furthermore, many languages avoid side-effects by limiting updates to only one per statement. C provides too many ways to update memory. These add nothing to the generality of the language, increase the opportunity for error, and complicate automatic optimisation. Restrictive practices are justifiable in order to accomplish correctly functioning and efficient software.

Java retains the ++ and -- operators, although with the removal of pointers and the addition of a decent string class, they are less necessary for idioms such as string and array manipulation. It is not clear whether they could cause side effects and subsequent problems as in C.

Eiffel has no such operators. They would merely be an unnecessary shorthand in Eiffel.

4.10 Defines

The define declaration -

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

has a different effect to -

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

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

#defines are poorly integrated with the language. The ‘#define’ must be in column 1, and is not subject to scope rules. Defines can lead to obscure errors, as the preprocessor does not detect them, but leaves them for the compiler. Programmers must be familiar with the particular preprocessor implementation on their system, as preprocessor implementations are different, particularly between Classic C and ANSI C.

#define also exhibits a multiple update problem:

```
#include <stdio.h>
#include <string.h>

#define dfn(x,y) ((x)<(y)?(x):(y))

main ()
{
    int i, j, k;

    k = dfn (i++, j);

    printf ("i = %d j = %d k = %d\n",
           i, j, k);

    i = 0; j = -1;
    k = dfn (i++, j);
```

```
printf ("i = %d j = %d k = %d\n",
       i, j, k);

i = 0; j = 5;
k = dfn (i++, j);

printf ("i = %d j = %d k = %d\n",
       i, j, k);
}
```

The results are as follows:

```
i = 1   j = 0   k = 0
i = 1   j = -1  k = -1
i = 2   j = 5   k = 1
```

This is even worse, if the actual parameter you pass is a function that updates other variables. All the variables will be updated the number of times the formal argument appears in the body of the define.

C++ at least reduces the need for defines by having inline functions. The problems with inlines have been discussed in their own section.

Java and Eiffel have no such preprocessing facilities. Where #defines are used as ‘cheap’ functions, ie., the code of the define is expanded inline in the invoking code, Eiffel and Java inline routines that meet certain criteria, without the side effects of #define.

#defines have often been used to provide a form of unrestricted genericity. In languages where genericity and templates are provided, this use for #defines disappears.

[Stroustrup 94] says he would like to see the preprocessor abolished: “The character and file orientation of the preprocessor is fundamentally at odds with a programming language designed around the notions of scopes, types, and interfaces.”

4.11 NULL vs 0

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

[Stroustrup 94] adds that “nothing seems to create more heat than a discussion of the proper way to express a pointer that doesn’t point to an object, the null pointer.” And, “The ARM further warns “Note that the null pointer need not be represented by the same bit pattern as the integer 0.”” Continuing on: “The warning reflects the common misapprehension that if p=0 assigns the null pointer to the pointer p, then the representation of the null pointer must be the same as the integer zero, that is, a bit pattern of all zeros. This is not so. C++ is sufficiently strongly typed that concept such as the null pointer can be represented in whichever way the implementation chooses, independently of how that

concept is represented in the source text.” No wonder people are confused, and there is much heated debate.

In Java `null` is a reserved word. Eiffel uses `Void`, the single value of type `NONE`, to indicate no object is referenced.

4.12 Case Sensitivity

It is good to adopt typographic conventions for names, which make a program more readable, but should not affect semantics. Distinguishing between upper and lower case in names can cause confusion, which leads to errors and systems that are difficult to maintain and modify. Case distinction is based on the implementation paradigm of how character codes work. Why do we have names? To give entities identity, and aid our memory of that identity. Philosophically, case distinction is contrary to the fundamental purpose of names, which introduces another form of overloading, the disambiguating mechanism being the underlying character codes.

Case distinction makes names harder to remember so is contrary to the purpose of a memory aid. Remembering command mnemonics or file names is difficult enough, let alone exactly the letter case. Your brain remembers the **sound** *fred*, not the characters used in spelling. In a case sensitive system, you must remember the letter case, whether it was fred, Fred or fREd, etc., greatly complicating the memory process.

Names are easier to remember than addresses. If we did not have names, we would have to retrieve files by addresses, access all machines on the Internet by their TCP address instead of host name, or call people by their social security number.

Case distinction in interactive systems is a poor user interface, being clumsy to continually use the shift key, which slows typing. Case sensitivity is one of the worst features of the Unix interface.

Consider the paradigm of letters and words. Words are spelt by assembling letters in order. There are 26 distinct letters. With the addition of digits 0 to 9, and the underscore character, we have a complete lexical definition for identifiers. Letters can be written in a number of styles. They can be bold, italic, upper or lower case. Such typographic representations, however, do not change the meaning of a word. Thus if we write ALGOL, Algol, **algol**, *Algol* (or maybe a star), we recognise the word to represent a computer language. The case of the letters or type style does not change the semantics.

Case distinction is based on the low level paradigm of character codes such as ASCII used internally in the computer. This weakens the purpose of using names to replace addresses, as names are reduced to a string of character codes.

Case distinction also contributes to errors, introducing ambiguity, which as has already been mentioned, weakens the purpose of names, as

identity is lost. As every programmer will have experienced, one character errors are more difficult to find than one would think. For example, if an identifier is declared Fred, another one can be declared fred, which are easily mistyped and confused.

We are generally poor proof-readers. The psychological reason for this is that the the brain tends to straighten out errors for our perception automatically. The human brain is an excellent instrument for working out what was intended, even in the presence of radical error. (This makes us good at difficult tasks like speech recognition.) Programmers must use their powers of concentration to override the natural tendency of the brain.

Case distinction adds cognitive difficulty. Good language design takes into account such psychological considerations in these small but important details, being designed towards the way humans work, not computers. Such considerations of cognitive science make a big difference to the effectiveness of people, but do not have any impact at all on the efficiency of code generated for the computer. What is more important, people or computers? With C the answer is often computers, as case distinction saves compiler processor cycles.

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

Another example of name overloading error is:

```
class obj
{
    int    Entry;

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

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

A common practice in C is to represent constants in upper case. This is actually bad practice, as a calling programmer should invoke a constant as a function that returns a value. The calling programmer does not need to know whether a class has implemented a feature as a constant, variable or value returning routine. This means that the class is free to change the implementation of the feature later, without having to bother all programmers to change the case of all occurrences of the identifier in order to follow some style rule.

It is amazing the passion that comes from those who defend case sensitivity. In fact, since I have argued for case insensitivity, some have said that this invalidates the whole of my critique of C++ because I don't agree with them on this point. The only point that is close to being valid for case sensitivity is that it forces all programmers to follow the same typographic convention for identifiers. This assumes that the burden of typographic considerations must be on programmers. I don't think it should be. This burden should be on the presentation medium, that is the editor or print formatter. For example, a program editor will know what an identifier is, and present it in lower case. Or it could even do this optionally, as some programmers might like to see identifiers in upper case, while others in lower case. This gives the best environment, where each programmer can tailor to their individual taste, and silly fights over style rules are forgotten.

Java has not improved this situation. In fact it is even worse, as Java uses Unicode instead of ASCII. The typographic form 'a' and 'a' could be different identifiers if one represents LATIN small letter a, and the other CYRILLIC small letter a. In Eiffel all words are case insensitive.

4.13 Assignment Operator

Using the mathematical equality symbol for the assignment operator is a poor choice of symbols; assignment is not equality ($:= \neq =$). Designers of ALGOL style languages realised they were semantically different, so took the care to distinguish, only using '=' in the sense of mathematical equality assertion. In C the confusion of notation leads to error, being easy to use = (assignment) where == (equality) is intended.

This leads to a more general criticism of C, in that it has a pseudo mathematical appearance. But then C is not very mathematical at all, as '=' does not represent equality, and C functions are not really functions. Few people are proficient at interpreting mathematical theorems, most passing over such sections in text, making the assumption that the mathematics proves the surrounding text. The pseudo-mathematical appearance of C is difficult to read, while lacking the semantic consistency and precision of mathematical notation. One of the keys of reusability is readability.

Java also uses the = symbol to mean assignment, so this has not improved. However, the = vs == confusion has been improved as in the syntax:

```
if ( Expression ) Statement
```

the Expression must have type boolean, or a compile-time error occurs.

Eiffel makes the clear distinction between the assignment operator choosing the ':=' symbol and mathematical equality '='.

4.14 char; signed and unsigned

What is the meaning of '+a', '-b', etc.; there is simply no real world equivalent. In C char, unsigned char, and signed char yield three distinct types all occupying 8 bits. These types are integers rather than characters. The definition is highly platform dependent, and the semantics is nonsense. Pascals technique of specifying integer subranges: 0..255, -127..+127, -63..+154, and so forth is far superior.

4.15 Semicolons

As with case sensitivity any discussion of this topic arouses passions that you wouldn't believe. Bjarne Stroustrup makes a very good observation on such debates: "Curiously enough, the volume of interest and public debate is often inversely proportional to the importance of a feature. The reason is that it is much easier to have a firm opinion on a minor feature than on a major one; minor features fit directly into the current state of affairs, whereas major ones - by definition - do not."

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

$$a + b +$$

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

In natural language both the comma and semicolon are separators, only the full stop is a terminator. If the comma were an expression terminator rather than separator, function invocations would look like:

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

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

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

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

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

```
"if" <condition> <statement>
"else" <statement>
```

In fact why do conditions in C if and while statements have to have parentheses around them? Why also must a semicolon follow the closing brace of a class, but must not follow the closing brace of a function?

Java being C based retains the semicolon as terminator. Eiffel views the semicolon as a separator, but has one advantage: semicolons are optional. The semicolon can be used to visually emphasise the separation between two commands, for example, where two commands are placed on one line.

4.16 Booleans

A serious omission from C was the boolean type. Booleans are fundamental to programming as conditions in **if..then** and loop constructs. C++ also has no built in boolean. It is interesting to see long Internet discussions on how booleans should be built, and how to represent the values, true and false. Using 0 to mean false, but any other value to mean true is unsatisfactory.

Java includes the basic type boolean, and so has rectified this situation. To accomplish C-style conversions you can use the expressions:

```
b = (i != 0);
i = (b)?1:0;
```

Eiffel takes a slightly different approach. As a language, Eiffel provides the mechanisms for building types. It has no assumptions about particular types built into the language. Types like *BOOLEAN* are defined as classes in the Eiffel Kernel Library, as are other basic types such as *INTEGER*, *REAL*, *STRING*, *ARRAY*, etc. This view is very similar to Smalltalk. These types are not built into the *language*, but they are usually built

into an Eiffel *compiler* so that there is no run-time performance penalty. This illustrates Eiffel's philosophy of keeping the language as small as possible, and as open as possible, so that programmers can build their own powerful types.

Recently the ANSI/ISO C++ committee has accepted `bool` as a distinct integral type. Before the definition of a boolean type in C/C++ could be any number of definitions which had slightly different semantics. If you were combining libraries that used these slightly different definitions, life could be difficult. This is probably a fundamental reason why libraries have not been as successful in C++ as they should be in an OO environment. Not all compiler implementations have implemented `bool` yet, so you can expect it to be years before this mess is cleaned up.

4.17 Comments

The following example comes from [GWS 94].

```
main ()
{
    int *i, *j;
    int k;

    k = *i/*j;
}
```

As they point out: what a good character combination `/*` was for delimiting comments.

4.18 Cpaghe++i

There are three kinds of *spaghetti* that occur in programs: *gotos*, *globals*, and *pointers*.

4.18.1 Cpaghe++i Gotos

Most people know about spaghetti *code* that is present in programs which use *gotos* in an undisciplined fashion. As Donald Knuth has pointed out it is entirely possible to produce well structured programs with *gotos*. The *well tempered goto* emulates high level structured statements such as conditionals, loops, switch or case statements in higher level languages.

Where a language provides the correct control structures, and the programmer programs into that paradigm, *gotos* are not needed. The reverse argument could also be made: if *gotos* cover all uses of high level control structures and even more, why have the high level control structures at all; why not just use *gotos*? The problem with *gotos* is that they are *too* powerful. They are too powerful in the same way assembler language is too powerful.

You can do everything with assembler or *gotos*, but it takes more work, and the result is often less than structured, difficult to understand and unmaintainable. The more work you do, the less efficient you are. It is not working harder that makes you more efficient, it is working smarter. I'm a great fan of laziness!

Consider what you must do to construct a loop with `gotos`: you must declare a label, then place the label and the `goto` somewhere; you also have to think about identifiers for labels that are non-ambiguous. For label identifiers, some languages use names, others numbers. With a high level loop construct, labels are implicit, meaning the programmer does not have this extra bookkeeping overhead. Then making changes becomes a lot more difficult, as you must create new labels, move them around, and delete others.

One legitimate use for `gotos` is to avoid overly complex nesting. Complex nesting usually occurs where there are many checks that result in multiply nesting `if...then`s, which often arise because of error checking. Proponents of `gotos` legitimately defend them for this situation. However, where the control structures are right, even this use of `gotos` is not needed.

Both Java and Eiffel abandon `gotos`. Java provides an extension to control structures which allows control structures to be named, and multi-level break and continue statements can be used to jump to an outer level conditional or loop.

In Eiffel the philosophy is to program in sufficiently small atomic routines, so that multi-level control structures are avoided. Thus Eiffel's solution to the nesting problem is integrated with its routine mechanism and the way programmers are expected to use routines. In object-oriented programming, it is good practice to keep routines small, with only one operation in a routine, as this enhances the possibility of reuse. Some programmers will object to small routines, as there is an overhead to routine calls, particularly in register based machines, where environments and registers must be saved. However, an Eiffel compiler will automatically inline small, non-polymorphic routines.

The high level language concept to remove the need for `gotos` altogether for error checking is exception handling. In this mechanism, the error condition triggers an exception. When an exception is raised, a search for its handler occurs. This search progresses down the run-time stack until an embedded exception handler is found. In Eiffel, exception handlers are specified in **rescue** clauses. Note that in an environment where exceptions can interrupt the flow of the code, garbage collection is even more important, as in a system with manual memory management, it is even more difficult to determine where to clean up, and which objects to dispose.

If exception raising and handling sounds expensive, then it should be realised that it often works out cheaper. Most of the time, the code runs normally, an exception being raised is the *exception*. Only then is the stack search for the handler performed. The mechanism actually works out cheaper in many cases. Consider divide by zero. In most systems, this exception is detected by the processor. If you don't have exception handling, you

must test that the divisor is not zero before a divide operation. With exception handling, you assume that the division will work in most cases, and so do not have to test. If the divisor is zero, you simply clean up in the exception handler. Only if there is no exception handler does the software fail.

The bottom line is that with the common high level language constructs of **if..then**, loops, cases, you can avoid *most* uses of `goto`. Add a high level construct for exception handling, and you can avoid `gotos` altogether.

4.18.2 Cpaghe++i Globals

The second kind of spaghetti is globals. Where two or more objects access the same set of globals, interdependencies arise between those objects. This makes it far more difficult to determine the correctness of a program, even more so in concurrent environments. These interdependencies should be viewed as strands of spaghetti worming their way through a system, which are going to make maintenance, extension, and reuse difficult in the future.

Globals can be abandoned. Objects are to globals as control structures are to `gotos`.

Again Java and Eiffel abandon globals, and thus ease the problems of maintenance, extension and reuse. Note that I use the word ease, not solve. Even though Java and Eiffel make significant improvements, there are no silver bullets to solve the problems involved in programming. Java and Eiffel are significant improvements.

4.18.3 Cpaghe++i Pointers

The third kind of spaghetti is pointers. The problems with pointer based programming are well known. The kind of spaghetti you get worming through the system is undisciplined pointers pointing to other elements, by-passing the whole concept of interfaces and object-orientation. Pointers introduce dependencies that would not otherwise be there. Furthermore, this can of worms results in dangling references and memory leaks. In order to do away with the problems of pointers, garbage collection is necessary. In order to implement good garbage collection pointers must be abandoned. C++ is caught in this *Catch-22*.

Neither Eiffel nor Java have pointers. Both have garbage collection built in from scratch.

While C++ overlays object-oriented concepts onto C, it is one of its greatest weaknesses that overlays OO on top of the spaghetti of a now old, low-level and flawed language. C++ does not enforce the advantages of the OO approach to remove these problems by programming only using published interfaces. The advantages of the OO paradigm are so effectively undermined in C++ as to be worse than useless. Many C programmers have thus stuck to C, and people like P.J. Plauger have been motivated to write papers such as "Programming Language Guessing Games: If C++ is the answer, what's the question?" [Plauger 93].

5. Conclusions

C++ is complex including too many constructs to overcome problems with itself and C, while lacking sophisticated mechanisms such as garbage collection, global analysis and automatic optimisations. C is thought of as being a simple language; but this is doubtful, as it has many operators, and a difficult precedence system. C's pointer style of programming is low level and difficult. Overall, C has many traps that lead to difficult to detect errors in software. Now C++ as a language is looking like the equivalent of computers of the 1950s, with large knobs, dials and patch panels; the C++ equivalents being pointers, structures, unions, #defines, etc., all of which have no place in a modern OO language, and are not in Java and Eiffel.

Compared to other OO languages, C++ looks more and more like an anachronism. C++ is now impeding the progress of the programming technology.

Object-oriented languages should provide sophisticated concepts in the simplest possible framework. In C++ the framework is not simple and the concepts are obscured. OOP addresses many issues in order to facilitate the production of complex and sophisticated programs. Many of these issues are addressed in implicit and subtle ways, but are lost in C++. Subtle errors can be introduced into C++ software in many ways; the combination of these causes further problems. C++ has devices for petty convenience, even the '+' itself, while sacrificing major conveniences, long-term correctness and safety, and the convenience of declarative programming, rather than operators. C++ forces the programmer to perform many administrative bookkeeping tasks that a compiler should automate.

It can be considered: what application domain is C++ relevant for? The answer to this is that C++ might be used as a better C. But for what applications is C relevant? C is relevant for low level Unix style programming, and is not an ideal language in view of its low level nature, and flaws. C is not applicable for large project organisation: hence C++'s attempt to improve it. C++, however, has not solved C's flaws, as I once hoped it would, but painfully magnified them.

Better languages exist for higher level functions such as communications and networks, scientific work, compilers, etc. I envisage that C has a place as a high level assembler that can be used to implement small pieces of code, where efficiency is of prime importance, on suitable platforms. Thus the use of C would be limited and well controlled, rather like small assembler routines are currently used in some systems. Indeed the move to C++ should only be considered in the case of upgrading a body of C programs for backwards compatibility. In the case of new projects alternatives to C and C++ should seriously be considered.

A programming language should embody the collective wisdom of common sense practices that have been learnt over many years, by common and painful experience. C++ does not implement much of this wisdom. [Sakkinen 92] observes that much of the C++ literature has few references to external work or research. It fails to draw on the insights and progress made by many researchers. This leads me to believe that C++ is parochial and removed from the many advances that will make production of systems easier and more cost effective.

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

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

A programming language cannot be seriously viewed as some authoritarian that stops us doing what we want or need to do. This view is still quite prevalent about languages with type safety and consistency checks.

This paper has shown many cases where C++ uses old C mechanisms to provide things that can and should be expressed consistently within the object-oriented paradigm. For example type casting. The move to pure object-oriented languages will facilitate more consistent programming and avoid many typical errors that occur in software production. C++ also makes distinctions that belong in the 'how' implementation domain. For example, '.' vs '->', and variables vs functions. These distinctions make bookkeeping work for programmers, which a compiler should handle. But then C++ fails to make distinctions that belong in the 'what' problem domain. For example, procedures vs functions. Making distinctions in the 'how' domain adds inconvenience to the language. Failing to make distinctions in the 'what' domain

limits the expressiveness of the language. The amount of change required in C++ to address the issues raised in this paper is seen as largely insurmountable, and Sun agrees with this.

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

As mentioned in the introduction, both sides of the analysis/design vs implementation debate need to compromise in order to bridge the semantic gap. The perpetuation of low level languages such as C into OOP is proof that the implementation community has not compromised, or sharpened its axe to bridge this costly gap. On the other hand the analysis/design community must realise that what they do is part of the general practice of programming.

It has been four years since the 2nd edition of this critique. The criticisms are still valid, but now many people have had first hand experience of being burnt by the OO hype and trying to implement systems in C++.

The work on languages such as Java and Eiffel has vindicated the criticisms previously made in the critique. [Stroustrup 94] lists as current C++ problems many of the criticisms I have also made in the critique. Java has recognised many shortcomings in C++ and rectified them. Many of the problems that Java fixes are the same problems as addressed in the original critique.

Eiffel serves as another example of better language design than C++. It has none of the problems of C++. In Java there still remain a few deficiencies, but it is a major advance.

Since the last edition of the critique, many people have asked what do I recommend. What should people choose then? Certainly Eiffel is the best out of these three languages. If you are doing large scale system software and application development, then the choice is Eiffel, although Eiffel is also simple and elegant enough for small

applications development. Eiffel is a language for the serious software engineer who wants to get on with the job, not be bogged down in syntactic and machine-oriented obscurities, weird 'bugs' and endless maintenance cycles to get things right.

Java is still an unproven entity for large projects, and the byte code is interpreted. Eiffel and C++ are roughly equivalent in performance. Interpreted Java will be around 10 times slower. But Java byte codes could be compiled into native code.

For small applets and other Internet loaded applications, Java is a good choice. Some people have predicated that Java will sweep all away, and that even Eiffel will die because of this. I cannot see this, as Eiffel and Java are really significantly different tools. Java has still to be tested in the large scale Eiffel league.

I have not yet mentioned languages such as BETA, Ada 95 or Smalltalk. BETA is still really in academia. It might make a stronger presence in the market place in the coming years. If not BETA might have the same profound influence as Simula. It is certainly something to be watched. Ada 95 is certainly aimed at serious software engineering.

Smalltalk is already firmly in the market place, and there are a significant number of systems that it is used for. Smalltalk is still a language for serious consideration. The biggest question here is do you want the development speed and flexibility of a dynamically typed system as opposed to the robustness and run-time speed of a statically typed system? Having answered these questions for yourself the choice between Smalltalk and Eiffel should be easier.

The most important aspect of C++ that the industry must realise is that the definition of C++ is unstable. As the X3J16 committee work on C++, more problems are uncovered. It will be years before a stable standard is reached, and probably years after that before compiler vendors are compliant with the standard.

Today's C++ programs will be tomorrow's unmaintainable legacy code. As [GWS 94] says of C++: "The seeds of software disasters for decades to come have already been planted and well fertilised." They compare C++ to COBOL in terms of unmaintainable legacy code which we have now in COBOL's case, and we will have in the future for C++.

Perhaps the most important realisation I had while developing this critique is that high level languages are more important to programming than object-orientation. That is, languages which have the attribute that they remove the burden of bookkeeping from the programmer to enhance maintainability and flexibility are more significant than languages which just add object-oriented features. While C++ adds object-orientation to C, it fails in the more important attribute of being high level. This greatly diminishes any benefits of the object-oriented paradigm.

In a nutshell, an object-oriented language that lacks the qualities of a high level language entirely misses the point of why we have progressed from machine coding to symbolic assembler and beyond. Without the essential high level qualities, OO is nothing but hype. Eiffel shows that it is important to be high level as well as OO, and I hope that the lesson to be learned by any programming paradigm, not just OO, is that the fundamental is to make the task of programming (that is system development as a whole) easier by the removal of the burden of bookkeeping.

C++ adds object-orientation to a low level language, so you still have all the bookkeeping burden of C. Java improves this situation by removing many of the low level features that have a known bad track record. Eiffel provides a true high level base for object-oriented programming.

The concluding advice of this critique is clear. Be wary of C++. Seriously consider the alternative languages.

Bjarne Stroustrup writes "My hope is that it will help C++ become accepted into areas that C failed to penetrate, and thus support programmers who have not been represented in the C and C++ culture." [Stroustrup 94] 6.5.3.1. My hope is that the industry establishes a professional software engineering culture, not a programming language culture based on seriously flawed and arcane languages. The software engineering culture is not well represented in C++.

Ian Joyner
October 1996

6. Bibliography

C++ ARM ELLIS and STROUSTRUP *The annotated C++ Reference Manual*, AT&T 1990.

[Adams 96] SCOTT ADAMS *The Dilbert Principle*, Harper Collins 1996.

[Aho 92] AHO and ULLMAN *Foundations of Computer Science*, Computer Science Press 1992.

[Brooks 95] FREDERICK P. BROOKS *The Mythical Man-Month*, 20th Anniversary Edition, Addison Wesley.

[Bruce 96] KIM B. BRUCE *Progress in Programming Languages*, in ACM Computing Surveys, Vol. 28, No. 1, March 1996.

[Capretz 87] PIERRE J. CAPRETZ *French in Action, A Beginning Course in Language and Culture*, Yale University Press.

[Cline] MARSHALL CLINE *C++ Frequently Asked Questions*, comp.lang.c++.newsgroup.

[DDH 72] DAHL, DIJKSTRA, HOARE *Structured Programming*.

[Deming 82] W. EDWARDS DEMING *Out of the Crisis*, Cambridge University Press 1982.

[Dijkstra 76] E.W. DIJKSTRA *A Discipline of Programming*, Prentice Hall 1976.

[DM&L 87] TOM DE MARCO and TIMOTHY LISTER, *Peopeware: Productive Projects and Teams*, Dorset House 1987.

[Ege 96] STUART HIRSHFIELD and RAIMUND K. EGE *Object-Oriented Programming*, In ACM Computing Surveys, Vol. 28, No. 1, March 1996.

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

[Flan 96] DAVID FLANAGAN *Java in a Nutshell*, O'Reilly & Associates 1996.

[GWS 94] GARFINKEL, WEISS, STRASSMANN *The Unix-Haters Handbook*, IDG books 1994.

[Hext 90] J.B. HEXT *Programming Structures: Machines and Programs. Volume I*, Prentice Hall of Australia 1990.

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

[Kilov and Ross 94] HAIM KILOV and JAMES ROSS, *Information Modelling: An Object-oriented Approach*, Prentice Hall 1994.

[L&S 95] WILLIAM J. LATZKO and DAVID M. SAUNDERS, *Four days with Dr. Deming: A strategy for modern methods of management*, Addison Wesley 1995.

[Madsen 93] MADSEN, MØLLER-PEDERSEN, NYGAARD, *Object-Oriented Programming in the BETA Programming Language*, Addison Wesley 1993.

[Meyer 88] BERTRAND MEYER *Object-oriented Software Construction*, Prentice Hall 1988. (2nd edition soon to appear.)

[Meyer 92] BERTRAND MEYER *Eiffel: The Language*, Prentice Hall 1992.

[Meyer 94] BERTRAND MEYER *Reusable Software: The Base Object-oriented Component Libraries*, Prentice Hall 1994.

[Meyer 95] BERTRAND MEYER *Object Success*, Prentice Hall 1995.

[Meyer 96a] BERTRAND MEYER *A Taxonomy of Inheritance*, IEEE Computer vol 29 No 5 May 1996.

[Meyer 96b] BERTRAND MEYER *Using Inheritance Well*, Chapter 25 of forthcoming *Object-oriented Software Construction*, 2nd edition Prentice Hall. Draft available on internet at <http://www.eiffel.com/doc/manuals/technology/oosc/inheritance-design/>

[Meyer 96c] BERTRAND MEYER *Concurrency, Distribution and the Internet*, Chapter 28 of forthcoming *Object-oriented Software Construction*,

2nd edition Prentice Hall. Draft available on internet at:

<http://www.eiffel.com/doc/manuals/technology/concurrency/CONCURRENCY.html>

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

[**Morgan 90**] CARROLL MORGAN *Programming from Specifications*, Prentice Hall 1990.

[**P&S 94**] JENS PALSBERG and MICHAEL I. SCHWARTZBACH *Object-oriented Type Systems*, Wiley 1994.

[**Plauger 93**] P.J. PLAUGER *Programming Language Guessing Games: If C++ is the Answer, what's the question?*, Dr Dobb's Journal, October 1993.

[**Reade 89**] CHRIS READE *Elements of Functional Programming*, Addison-Wesley, 1989.

[**RBPEL91**] RUMBAUGH, BLAHA, PREMERLANI, EDDY, LORENSEN *Object-Oriented modelling and Design*, Prentice-Hall, 1991.

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

[**Shaw 96**] MARY SHAW and DAVID GARLAN *Software Architecture: Perspectives on an emerging discipline*, Prentice Hall 1996.

[**SJE 91**] SAAKE, JUNGCLAUS, EHRICH *Object-Oriented Specification and Stepwise Refinement*, in IFIP Workshop on Open Distributed Processing Berlin, 1991.

[**Stroustrup 94**] BJARNE STROUSTRUP *The Design and Evolution of C++*, Addison Wesley 1994.

[**Sun 95**] *The Java Language Environment: A White Paper*, Sun 1995. (<http://java.sun.com>)

[**Sun 96**] *The Java Language Specification*, Sun 1996. See WEB address.

[**Weg 91**] PETER WEGNER *Concepts and Paradigms of Object-Oriented Programming*, ACM SIGPLAN OOPS Messenger Volume 1 no. 1 August 1990.

[**Wiener 95**] RICHARD WIENER *Software Development Using Eiffel: There can be life other than C++*, Prentice Hall 1995.

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

[**Yoshida 92**] KOICHIRO YOSHIDA Title and book in Japanese.

7. Bibliography

Ada 95:

Home page

<http://lglwww.epfl.ch/Ada/>

Ada 95 Guide for C/C++ Programmers

<http://lglwww.epfl.ch/Ada/Ammo/>

Cplpl2Ada.html

Contrast to C++ by Edmond Schonberg

<http://www.csci.unt.edu/faculty/ryan/languages/ada/9x-cplus.txt>

Beta:

<http://www.daimi.aau.dk/~beta/>

C++:

FAQ

<http://www.cs.bham.ac.uk/~jdm/Cpp/cppfaq.html>

ISO SC22/WG21 standards

<ftp://research.att.com/dist/c++std/WP>

<ftp://ftp.maths.warwick.ac.uk/pub/c++/std/WP>

<http://www.cygnus.com/misc/wp/index.html>

http://reality.sgi.com/employees/austern_mti/std-c++/faq.html#B8

STL

<http://www.cs.rpi.edu/~musser/stl.html>

Comments on Critique

<http://www.cs.oberlin.edu/students/jbasney/critique/critique.html>

Dilbert:

The Dilbert Zone

<http://www.unitedmedia.com/comics/dilbert/>

Eiffel:

EiffelWorld magazine

<http://www.eiffel.com/doc/eiffelworld/>

Downline load site for SmallEiffel

<ftp://ftp.loria.fr/pub/loria/genielog/SmallEiffel/>

Interactive Software Engineering

<http://www.eiffel.com/>

Dynamic Linking in Eiffel

<http://www.eiffel.com/doc/manuals/dle/book>

Vendor independent home page

<http://arachnid.cs.cf.ac.uk/CLE/>

Books on Eiffel

<http://www.eiffel.com/doc/documentation.html>

SIG computer and Visual Eiffel

<http://www.sigco.com/>

Tower Technology

<http://www.twr.com/>

Eiffel locater page

<http://www.progsoc.uts.edu.au/~geldridg/stop-press.html>

Java:

Main page

<http://java.sun.com/>

Demonstration Applets

<http://www.gamelan.com/index.shtml>

Java Language Specification

http://java.sun.com/doc/language_specification/index.html

Oberon:

The Oberon Reference Site

<http://www.math.tau.ac.il/~laden/oberon/>

Sakkinen, Markku:

<http://www.cs.jyu.fi/~sakkinen/>

References to other papers on C++ and other topics by Dr. Sakkinen.

X3J16 C++ ISO standardisation:

http://www.x3.org/tc_home/x3j16.html