1. Design by Contract

1.1 Intoduction

This chapter considers the development of the notion of Design by Contract. It looks at the origin of the idea within Computer Science, and considers how this compares with the notion of contract in everyday affairs. Mention is made of attempts to support Design by Contract in various programming languages.

A detailed look is taken at the Eiffel language, which is synonymous with the methodology in the minds of many people. A number of significant aspects are discussed, including the various types of assertion that can be used to express a software contract, the capability of assertion-based software to be self-documenting, the limitations of the Eiffel assertion language (and the dangers of fabricating extensions), and the use of exceptions to handle the incidents when assertions are not satisfied. It is necessary to precede this with a brief treatment of the notion of subtyping in languages that support inheritance, looking at covariance and contravariance and at the Liskov Substitutability Principle.

The bulk of the chapter looks at attempts to support Design by Contract in C++. In each case, these attempts can be classified as one of the following approaches:

- use the existing language constructs to provide a support mechanism;
- use macros to support the inclusion and monitoring of constraints;
- effectively extend the syntax of the language by writing a pre-processor which converts non-C++ constraint expressions into C++ prior to compilation;
- propose a language extension.

By taking a thorough overview of previous work, it is possible to derive a fairly comprehensive list of desirable features for a support mechanism for Design by Contract in C++ which can inform future work, and indeed the developments documented in the following chapter. By the same token, a list of potential problems and pitfalls can also be compiled. Both of these are presented at the end of the chapter, and provide a yardstick by which the efficacy of future developments can be measured.

1.2 Origin of Design by Contract

According to Tony Hoare [i]:

"An early advocate of using assertions in programming was none other than Alan Turing himself. On 24 June 1950 at a conference in Cambridge, he gave a short talk entitled "Checking a Large Routine" which explains the idea with great clarity. "How can one check a large routine in the sense that it's right? In order that the man who checks may not have too difficult a task, the programmer should make a number of definite assertions which can be checked individually, and from which the correctness of the whole program easily follows."

The notion of assertions was further developed by Hoare [ii, iii], Floyd [iv] and Dijkstra [v]. Unfortunately, commercial programming languages rarely provide support for assertions, let alone the extensions (e.g. inheritance) needed for supporting Design By Contract. Some attempts to remedy this are noted below.

1.2.1 The Notion of Contract

Human contracts involving two parties are characterised by two major properties:

- each party is persuaded to enter into the contract by the benefits on offer; they accept that along with priviledge comes responsibility, and are willing to therefore incur some obligations to obtain the benefits;
- these benefits and obligations are documented in a contract document. The contract document protects both parties. It protects the client by specifying the minimum that should be done: the client is entitled to receive a certain result. It protects the supplier by specifying how little is acceptable: the contractor must not be liable for failing to carry out tasks outside of the specified scope.

Clearly, what is an obligation for one party involved is usually a benefit for the other.

There is further point worth noting about human contracts which will bear comparison with the software contracts discussed later. The contract obeys the No Hidden Clauses rule. That is, with a fully spelled out contract between two honest parties, no requirement other than the contract's official obligations may be imposed on the client as a condition for obtaining the contract's official benefits: the client who satisfies their obligations is automatically entitled to the benefits. At the same time, the rule does not exclude the inclusion of references (implicitly or explicitly) to rules that are not physically part of the contract, such as the relevant laws and standard practice for the profession in question. The situation whereby every undertaking in a particular context is subject to the same contract gives rise to the notion of a class invariant when it comes to software.

1.3 Design by Contract in Languages other than C++

1.3.1 Non Object-Oriented Languages

1.3.1.1 Algol W

Wirth and Hoare's Algol variant provided support for executable assertions. It was not widely used and Pascal, Algol's direct successor, dropped support for assertions. Since Pascal was intended to be used to teach programming, this begs the question what sort of programming it was intended to teach.

1.3.1.2 Ada

ANNA [vi] – ANNotated Ada – was an attempt to append assertions to the definition of an existing programming language which was fairly widely used (at least in the US Department of Defence). It was based on Ada83, not the object-oriented Ada95. As such, its assertions were limited to data transformation rather than the more abstract level required for Design By Contract. Nonetheless, it has been used to help bridge the gap between formal specifications written in Z and an actual Ada implementation.

It is unfortunate that ANNA was not incorporated into Ada95, where there would have been more widespread opportunity for the value of Design By Contract to become recognised by a greater audience.

1.3.2 Object-Oriented Languages

1.3.2.1 Liskov Substitutability Principle

In order to consider properly supporting Design by Contract across an inheritance hierarchy, it is important to understand the notion of subtype (as opposed to simply subclass). The rules that are derived from this notion are known as the Liskov Substitutability Principle [vii]. In order for one type to be a subtype of another, objects of the subtype must behave the same as those of the supertype as far as anyone or any program using supertype objects can tell.

In Eiffel, if the following entities are declared:

```
p: POLYGON; r: RECTANGLE; t: TRIANGLE
```

and the classes RECTANGLE and TRIANGLE inherit from POLYGON, then the following assignments are valid:

```
p:=r
p:=t
```

After the assignment, p will behave according to its apparent type POLYGON, and it is expected that if the program performs correctly when the actual type of p's object is a POLYGON, then it will also work correctly when the actual type of p's object is a subtype of POLYGON. The Liskov Substitutability Principle defines the requirements that guarantee this behaviour. In particular, redefined methods must not exhibit different behaviour (such as a dateOfBirth() function being redefined to return the current date instead) and must not refuse to handle any of the cases that the original handled (such as an operation being redefined to exclude a degenerative case).

1.3.2.1.1 Covariance and Contravariance

The terms covariance and contravariance refer to the issue of type conformance for the signatures of redefined methods of subtypes. Originally, Eiffel adopted covariance for all types involved: a feature redeclaration could change either (or both) the result type or argument types of the signature as long the new types of both the result and the arguments conform to the originals. (One type conforms to another if it is derived from it by inheritance).

When William Cook [viii] reported that there were loopholes in Eiffel's type system, this came as a surprise to those who look to Eiffel as a model for strongly-typed object-oriented languages. Cook's amendments were intended to enforce strict subtyping; notable amongst them was inverting the routine argument redefinition rule to observe contravariance (redefined arguments should have more general types, so that the original argument conforms to the new), whilst retaining covariance for redefined results (a redefined result must be subtype). These rules ensure that objects of type Y can safely be passed to variables of type X when Y is a subtype of X.

Different languages take other approaches. For example, C++ adopts a novariance policy to argument types. Any change in the arguments of a routine redefined in a subtype constitutes a different routine. The same does not apply to return types, however: return types can be changed, and not necessarily covariantly. Sather, on the other hand, covariantly types the 'hidden' parameter to the bound object and the return type, and contravariantly types all routine arguments.

However, even when the covariance and contravariance rules are correctly applied, they are not strong enough to ensure that the program containing the above assignment will work correctly for any subtype of POLYGON, since these rules only ensure that no type errors will occur. Like type checking, covariance and contravariance capture only a small part of what it means for a program to be correct. For example, both a STACK[T] class and a QUEUE[T] class might provide operations

```
void put(T)
T get(T)
```

which add and remove items. Under the contravariance rule STACK[T] could be a subclass of QUEUE[T] and vice versa. However, applying these two operations to an object in immediate succession is almost certain to yield different results depending on whether the object to which they are applied is in fact a STACK[T] or a QUEUE[T].

What is required in order to pin down a subtype is a stronger requirement that constrains the behaviour of subtypes: properties that can be proved using the specification of an object's presumed type should be observed to hold when the object in question is actually an instance of a subtype of the presumed type. Thus the *SubType Requirement* is stated by Liskov as:

Let $f(x)$ be a property provable about objects x of type T. Then $f(y)$ should be true for objects y of type S where S is a subtype of T.

It needs to be understood that this definition refers only to *safety* properties: those which state that "nothing bad happens"; the scope of the definition does not include *liveness* properties (those which state that "something good eventually happens").

An abstract data type is defined in terms of its properties. There are four sections to the formal specification of an abstract data type: TYPES, FUNCTIONS, AXIOMS, PRECONDITIONS. Type checking and application of covariance/contravariance only ensures that the properties described in the first two of these sections are maintained. In order for a type to properly be subtype of another (as opposed to just derived from it), the properties described in the other two sections must be maintained also. Thus if there is an invariant on the type, the subtype must uphold that invariant. If there is a postcondition on a function of the type, and the subtype redefines the function, it must still guarantee the postcondition. And if a function of the type has a precondition defining the domain of the function, a redefinition of that function in the subtype must act on at least that same domain.

The paper by Liskov and Wing dealing with this subject introduces an interesting additional slant to the argument, in that they include history properties, an aspect not dealt with in any other literature considered. Whilst constraints usually deal with properties that are true in a particular state (the current state of the current object or objects), history constraints are properties that are true of all sequences of states. Thus, invariants are formulated as predicates over single states, and history properties are formulated as predicates over pairs of states. For example, an invariant property of a bag is that its size is always less than its lower bound; a history property is that the bag's bound does not change.

1.3.2.2 Eiffel

In the Eiffel [ix] language, the notions of human contract described earlier are modelled by assertions. Eiffel provides the most comprehensive support for assertions in an object-oriented language, and are the foundation of Meyer's Design By Contract [x] method. If the execution of a certain task relies on a routine call to handle one of its subtasks, it necessary to specify the contract between the client (the caller) and the supplier (the called routine) as precisely as possible. Assertions are the mechanism for expressing the terms of the contract.

Meyer makes the point that it is important that the terms of the contract are included as part of the definition of the routines which undertake the work of the contract. The contract contains the most important information that can be given about the routine: what each party in the contract must guarantee for a correct call, and what each party is entitled to in return. He argues that because this information is so crucial to the construction of reliable systems that use such routines, it should be a formal part of the routine's text. The argument is compelling, but it should be noted that this facility can only be enjoyed in a language that supports it.

1.3.2.2.1 Preconditions and Postconditions

Each routine in Eiffel can state a postcondition which is a predicate about the state of the system after the routine has completed execution. The property expressed by the postcondition is assumed to be desirable: clients are expected to want to enter into contracts with that routine (the supplier) in order to enjoy the benefits of the contract. Counterbalancing this, each routine can also state a precondition, which is a predicate about the state of the system before the routine has begun execution. Only if the system is in a state satisfying the precondition predicate is the postcondition's predicate guaranteed at the end of the routine. An example of a class feature with a precondition and postcondition is shown below.

```
class STACK[T]
...
put(x: T) is
require
  not_full: not full
do
  count := count+1
  representation.put(count, x)
ensure
  not_empty: not empty
  added_to_top: item=x
  one_more_item: count = old count + 1
end
```

This example illustrates some of the aspects of Eiffel's assertion mechanism. Firstly, the assertions are (optionally) labelled. This label helps the programmer quickly identify the nature of the violation when a constraint is not satisfied. Secondly, the assertions in the postcondition appear on separate lines so that they can be individually identified (by their label). The complete postcondition (or precondition) is the conjunction of these assertions. Thirdly, the assertions use features (call functions) defined elsewhere in the class, such as `empty` and `item`. This has implications when considering program behaviour, which are dealt with later. Fourthly, Eiffel provides the postcondition with access to the value a data item held when the routine began through the **`old`** mechanism.

1.3.2.2.2 Eliminating Redundant Checking

Note that it is completely contrary to the methodology for the body of the routine to actually check that the precondition is satisfied before proceeding. One of the intentions of Design by Contract is that code clutter is reduced in routine bodies since they do not have to check the precondition: it is responsibility of the caller to ensure that the precondition is met. Checking the precondition in the routine only appears to make the routine more robust: in fact, software quality degenerates because of the increased amount of code (and therefore complexity, and therefore scope for error) and execution suffers because of the redundant checking.

One objection to this style is that it seems to force every client to make checks corresponding to the precondition, which simply relocates the clutter. However, this argument is not justified. The presence of a precondition p in a routine r does not necessarily mean that every call of r must test for p, as in

```
if x.p then
  x.r
else
  -- special treatment
end
```

What the precondition means is that the client must guarantee property p; this is not the same as testing for this condition before each call. If the context of the call implies p, then there is no need for such a test. Suppose that the following two statements occurred in the course of the program code.

```
x.s; x.r
```

Then if the postcondition of routine s implies the precondition of routine r, there is no need to check that r's precondition is satisfied before calling r.

1.3.2.2.3 Invariants

Eiffel permits the stating of a class invariant which is characterised by two properties: firstly, the invariant must be satisfied after the creation of every instance of the class, so every creation procedure must yield an object satisfying the invariant; and secondly, the invariant must be preserved by every exported routine of the class (i.e. every routine which is available to clients of the class). Any such routine must guarantee that the invariant is satisfied when the routine completes execution (provided that it was satisfied on entry).

The life cycle of an object can therefore be pictured as a sequence of transitions between "observable" states. "Observable" here means states that are visible to the client: the state of an object after creation, and after the action of exported routines. The invariant does not have to be satisfied by features that are not exported. Applying these features may form part of the execution of visible routines, and the object may be temporarily in an inconsistent state after their execution, but as long as consistency is restored before the completion of the routine, the invariant is not considered to have been violated: the invariant is the consistency constraint on the observable states only.

1.3.2.2.4 Assertions with Side Effects

The assertion language in Eiffel does not cover all expressions of first order predicate calculus. The assertions are Boolean assertions, with a few extensions such as the **old** notation for denoting the value of a data item at the commencement of routine execution. Universal and existential quantification are not provided, but they can be expressed by writing functions that contain loops that emulate the quantifiers. In fact, first order predicate calculus would not be sufficient even if it were supported in full within in the language: in practice, many properties require higher-order calculus for their expression (e.g. determining whether a graph is cyclic).

The use of functions within assertions is not without its dangers, since there is a significant difference between Boolean predicates with and without them. Functions are computational, they have the capacity to alter the state of the system: they are imperative. Mathematical functions, on the other hand, are applicative: there are no side-effects. The difference (and apparent lack of it) is clear in the following definition of a STACK[T] feature which checks if the stack is full:

```
full: BOOLEAN is
do
   Result := (count=capacity)
ensure
   full_definition: Result = (count=capacity)
end
```

The instruction following **`do`** is a command given to the virtual machine on which the program runs which causes its state to change. The assertion following **`ensure`** does not do anything: it simply specifies an expected property of the state after routine execution.

As Meyer puts it, 'introducing functions into assertions lets the imperative fox back into the applicative chicken coop'. In practice, this means that there is an obligation on the assertion writer to only use functions of impeccable character, which it can be verified will not affect the program state.

1.3.2.2.5 Assertions as Documentation

In order for clients to make correct use of a class and its features, they must be clear about what it does: its semantics. The *short* command in Eiffel documents a class by extracting interface information. This information is essentially the implementation-independent part of the class text, and therefore the class documentation is part of the class, not a document developed and maintained separately from it. The *short* command retains only the exported features of the class, along with the preconditions, postconditions and invariants.

For derived classes, another tool is required to supplement the *short* command, called the *flat* command, which ensures that inherited features are included in the short form of the class alongside those defined in the class itself.

1.3.2.2.6 Assertions under Inheritance

The fact that one class can be a derivative of another provides the opportunity for polymorphism: an entity of a declared type can in fact refer at different times to objects of many different types, as long as the alternative type is derived from the original declared type. Dynamic binding ensures that a routine call picks up the definition of the routine that is appropriate to the actual type of the object to which the entity refers, not that which is indicated by the type of the entity declaration. Inheritance also provides the opportunity to redefine the action of a routine. Taking all these features together provides an extremely powerful and flexible mechanism, but one which can be abused: there is the potential for the semantics of the original routine declaration to be changed in a derived class, and then for an entity of the type of the original class to display those changed semantics.

In order to guard against this, Eiffel uses a simple convention. In a reclaration, it is not permitted to use the forms **`require`** and **`ensure`**. This means that the original precondition and postcondition are retained, and can only be augmented by the use of **`require else`** and **`ensure then`**. Either augmentation has the effect that the subcontractor does a better job than the original contractor. Weakening the precondition means that the routine offers its benefits for less obligation; strengthening the postcondition means that the benefits it offers are more desirable. These amendments yield the following as a new precondition or postcondition:

```
new_precondition or else original_precondition
new_postcondition and then original_postcondition
```

The Eiffel boolean operators **`and then`** and **`or else`** (whose names are borrowed from Ada) are non-strict. This means that it is not necessarily the case that both operands are evaluated. If the truth or falsehood of the expression can be determined by evaluation of the first operand, the second is not evaluated. Therefore, if the second operand is undefined, the expression may still have a defined value. Clearly, the non-strict operator is therefore not commutative.

Thus, in this specific case, the original precondition is not evaluated if the new precondition is satisfied, and likewise there is no need for the original postcondition to be evaluated if the new postcondition is not satisfied. In addition, invariants are inherited, so that the effective invariant of a derived class is the conjunction of its stated invariant with the (effective) invariants of any classes it inherits.

1.3.2.2.7 Handling failed assertions in Eiffel.

When an assertion is violated, Eiffel provides a disciplined exception-handling mechanism. There is a hierarchy of exception handlers defined, with a default handler at the top level (which informs the user of the exception and terminates the program) to ensure that all exceptions are handled. When an exception occurs, it is propagated first to the caller, and if the caller defines an exception handler – known as a 'rescue' clause – control is passed to it. The caller then has the opportunity to attempt to recover and overcome the problem, perhaps by executing an alternative strategy. If this is not possible, at least the opportunity to exit gracefully, tidying up the system state, is available. If the caller does not define an exception handler, the exception is passed up the function call stack until a handler is encountered.

Eiffel provides a multi-level debugging mechanism by offering fine granularity of control over assertion enabling: each of the different types of assertion statement can be individually enabled. When a program is first created, every assertion can be enabled. When the developer or tester is satisfied that the code is correct, some or all of the exceptions can be disabled to optimise execution speed. Typically, the release version of a class has only the preconditions enabled since the internal implementation of the class has been verified through testing. This is particularly true of library classes. Leaving preconditions enabled not only highlights improper use of the library, it can also act as a debugging aid for client classes of the library: if the client writer believes (mistakenly) that their software creates a state satisfying the precondition of a library routine, the library class is able to highlight when this is not the case.

1.3.2.2.8 Abstract Preconditions

In Object-Oriented Software Construction (2nd Edition) Meyer describes what he calls 'abstract preconditions'. For example, a Stack[G] class has a put(x : G) feature for which the precondition is given as '**`not`** `full`'. This refers to the class feature `full`, which can be redeclared in derived classes. Thus a descendant like

BoundedStack[G] can define override the default of full() (Result:= false) and effectively strengthen the precondition. He argues that the abstract precondition is unaltered, and only its concrete representation changes. This is a debatable point, on which there seems to have been little or no theoretical work.

1.3.2.3 Sather

Sather [xi] is a direct descendant of Eiffel. It is a research project at the UCB International Computer Science Institute (ICSI) on object-oriented language design for high-performance parallel computing. Below is an example of assertions 'pre' and 'post' used in Sather:

```
class CALCULATOR is
  readonly attr sum: INT; -- Always kept positive.
  add_positive (x: INT): INT pre x > 0 post result >= initial(x)
  is
  return sum + x;
end
```

Despite the influence of Eiffel, assertions were an after-thought in Sather. They were not in the initial version and later versions lack support for assertions in abstract classes. Even the documentation describes assertions as "safety features for the earnest programmer to annotate the intention of code". This hardly recognizes their role in Design By Contract.

1.3.2.4 Java

Perhaps the latest development in Design By Contract is the iContract tool for Java [xii]. Since the Java language provides even less support for Design By Contract than C (there are no optional assertions), the approach used here is to apply a pre-processor that recognizes stylized comments indicating preconditions, postconditions and invariants, and generates instrumented Java source code that checks the various kinds of assertions.

1.4 Supporting Design by Contract in C++

The number of efforts that have been made to provide support for Design by Contract in C++ are testimony to the how widespread is the belief that such support would offer significant benefits for software quality.

In theory, the most straightforward approach to supporting Design by Contract in C++ is to use Eiffel for those components for which monitoring constraints is most important. However, since this requires familiarity with two different languages and compilers and the ability to integrate external routines, not to mention finding a way to handle cross-language exception handling, it is not a practical option for the majority of C++ users.

There have been a number of other approaches to supporting Design by Contract in C++: use the existing language constructs to provide a support mechanism; use macros to support the inclusion and monitoring of constraints; effectively extend the syntax of the language by writing a pre-processor which converts non-C++ constraint expressions into C++ before compilation; and proposing language extensions.

1.4.1 Using Existing Language Constructs

The Percolation pattern [xiii] uses existing language constructs in a straightforward structure that yields the same checking as built-in assertion inheritance. It makes use of the Gang of Four Template Method [xiv], which lets subclasses redefine certain steps of an algorithm without changing the algorithms structure. The Percolation pattern relies on the redefinitions observing a protocol in which they incorporate calls to base class functions and observe the Liskov Substitutability Principle in doing so. The pattern requires discipline on the part of the programmer, which can have implications for maintenance.

1.4.1.1 The Percolation Pattern

The description of the pattern provides a very thorough (some might say over-laborious) consideration of the issues surrounding the need to observe the LSP. However, it does not offer any assistance in automating the process: it simply lays down the protocol that must be observed by the programmer.

The treatment identifies three cases of method preconditions and postconditions which differ on how inheritance has been used to compose the method. The first case is simply the base class (i.e. inheritance has not been used). In such cases, the preconditions and postconditions of the routines are as stated, after their conjunction with the class invariant has been evaluated. In the second case, the class is composed by inheritance but does not provide a redefinition of the method under consideration. In this case, the preconditions and postconditions of the superclass are used together with the conjunction of the class invariant with the superclass invariant. In the third case, the method is overwritten (redefined). The new precondition is the disjunction of the class precondition and the (resultant) precondition of the superclass (and of course the conjunction with the class invariant). 'Resultant' refers to the complete precondition of the class, including any inherited disjunctions. The new postcondition is the conjunction of the class postcondition and the (resultant) postcondition of the superclass (and again the conjunction with the class invariant).

In fact, the first two cases are simply special cases of the third, and it is only necessary to consider a single example, in which the action for the other cases is evident. A method `foo` is shown, in which the placement of the assertions follows these recommendations: invariant after constructor body, invariant and precondition before method body, invariant and postcondition after method body, invariant before the destructor body. Only a single constructor is shown – the same structure would be implemented for all constructors.

Each precondition, postcondition, and invariant is implemented as a protected member function. This makes the superclass assertions visible to subclasses without exposing them to clients. These member functions are all type bool (returning true when the assertion holds, false when it is violated), inline (to increase performance and then allow the optimizer to remove the entire function when they are null), and `const` to ensure there are no side effects. The assertion function is implemented with `ASSERT` statements (`ASSERT` is an altered assert which outputs the string in the assertion: program termination would obviously mean that the return value of the preconditions and postconditions would never get to be returned as false). When the assertion expansion is disabled, an empty inline function results. Empty inline functions are completely removed by the compiler's optimization.

The application method must make two calls: first to the invariant function, then to the pre or post function. If the invariant call was packaged with the pre or post function, then superclass invariants would be called twice (once when the invariant is percolated and a second time when the precondition is percolated.)

```
Class Base {
public:
  Base() { invariant(); }

  virtual void foo() {
    invariant();
    fooPre();

    // foo body
    invariant();
    fooPost();
    return;
  };

  virtual ~Base() { invariant(); }

protected:
  inline bool invariant() const {
    ASSERT(baseValue == requiredBaseValue, Base Invariant");
  };

  inline bool fooPre() const {
    ASSERT(fooValue == requiredfooValue,  "Base::foo Precondition");
  };

  inline bool fooPost() const {
    ASSERT(fooValue == requiredfooValue,  "Base::foo Postcondition");
   };
};

Class Derived1: public Base {
public:
  Derived1 () { invariant(); }

   virtual void foo()  {
    invariant();
    fooPre();

    // foo body
    invariant();
    fooPost();
    return;
  };
```

```
  virtual ~Derived1() { invariant(); }

protected:
  inline bool invariant() const {
      ASSERT((someD1_Value == required_d1_Value && Base::invariant()),  "Derived1
Invariant");

  };
  inline bool fooPre()  const {
      ASSERT((fooD1Value == requiredfooD1Value || Base::fooPre()), "Derived1::foo
Precondition");

  };
  inline bool fooPost()  const {
    ASSERT((fooD1Value     ==     requiredfooD1Value     &&     Base::fooPost()),
"Derived1::foo Postcondition");
  };
};

Class Derived2 : public Derived1{
public:
  Derived2 () { invariant(); }

  virtual void foo()  {
    invariant();
    fooPre();

    // foo body
    invariant();
    fooPost();
    return;
  };

  virtual ~Derived2() { invariant(); }

protected:
  inline bool invariant() const {
     ASSERT((someD2_Value   ==   required_d2_Value   &&   Derived1::invariant()),
"Derived2 Invariant");

  };
  inline bool fooPre()  const {
    ASSERT((fooD2Value    ==    requiredfooD2Value    ||    Derived1::fooPre()),
"Derived2::foo Precondition");

  };
  inline bool fooPost()  const {
    ASSERT((fooD2Value    ==    requiredfooD2Value    &&    Derived1::fooPost()),
"Derived2::foo Postcondition");
  };
};
```

Clearly the mechanism is very verbose, and there is a large amount of repetitive coding required. Hence there is plenty of scope for error, especially in maintenance. The repetition takes a number of forms:

(i) supporting a second method `fee()` would require the addition of practically as much code as is already shown: only the definitions of `invariant()`, and of the constructors and destructors, would not require additional work on the part of the programmer.

(ii) the example given does not show method parameters. There would also be a lot of repetition involved in including method parameters, since these must be made available to both the precondition and the postcondition. The method parameters would therefore need to be repeated four times in the Base (method body definition and condition method definition for both precondition and postcondition), and six times in each Derived class (additional call to superclass).

(iii) for each method `foo(..)`, additional names `fooPre(..)` and `fooPost(..)` are introduced into the class scope.

(iv) if there are multiple return statements in a method `foo(..)`, either the code of the method must be reworked to use a single return statement preceeded by the calls to `invariant()` and `fooPost(..)`, or these calls must be repeated preceeding every return statement.

1.4.2 Using a Pre-Processor or Compiler Front End

1.4.2.1 App

App [xv] is an Annotation PreProcessor for C programs developed in UNIX environments, inspired by ANNA. App recognises assertions that appear as annotations of the source text, written using extended comment indicators (/*@...@*/). It is designed on the premise that errors most often occur at function interfaces, and is intended primarily to monitor constraints at such points: it recognises three assertion constructs in this respect: **assume** (preconditions), **promise** (postconditions), **return** (specifies a constraint on the return value of a function), plus a fourth construct, **assert**, which specifies a constraint on a intermediate state of a function body. To discourage writing assertion expressions that have side effects, App disallows the use of C's assignment, increment and decrement operators in assertion expressions. Of course, functions that produce side effects can be invoked within assertion expressions, and App cannot prevent this.

App also provides three constructs additional to C syntax to enhance the expressivity of constraints: **all**, **some** and **in** provide universal quantification, existential quantification and reference to the value of a variable prior to routine execution respectively.

An example of the use of these constructs given by Rosenblum is reproduced below.

```
int* sort(int *x, int size)
/*@
  assume x && size>0;
  return S where S  // S is non-null
    && all (int i=0; i < in size-1; i++) S[i] <= S[i+1]  // S is ordered
    && all (int i=0; i < in size; i++)
      some (int j=0; j < in size; j++)
        x[i]==S[j];  // S is a permutation of x
@*/
{
// sort body
}
```

This illustrates the idea, although it is unfortunate that the stated conditions are not in fact correct. Consider for example the initial value represented by x=[1, 2, 2, 3], and the possible return value S=[1, 2, 3, 3]. Then S fulfils the conditions stated by the **`return`** constraint, but is not a permutation of x. The desired condition requires the definition of an auxiliary function such as that below

```
bool perm(int *a, int *b, int size_a, int size_b)
{
  if (size_a!=size_b)
    return false;
  else
    if (size_a==1)
      return a[0]==b[0];
    else
      for (int i=0; i<size_b)
        if (b[i]==a[0])
          return perm(a+1, delete(b, i, size_b), size_a-1, size_b-1);
  return false;
}

int *delete(int *x, int i, int size_x)
{
  for (int j=i; j<size_x-1; j++)
    x[j]=x[j+1]
  return x;
}
```

so that sort becomes

```
int* sort(int *x, int size)
/*@
  assume x && size>0;
  return S where S  // S is non-null
    && all (int i=0; i < in size-1; i++) S[i] <= S[i+1]  // S is ordered
    && perm(S, x, size, size)  // S is a permutation of x
@*/
{
// sort body
}
```

App converts each assertion to a runtime check, which tests for the violation of the constraint specified in the assertion. If the check fails at runtime, then additional code generated with the check is executed in response to the failure. The default response code generated by App prints out a simple diagnostic message, sufficient to indicate which assertion was violated. However, the response to a violated assertion can be customised to provide diagnostic information that is unique to the context of the assertion. This is accomplished by attaching a violation action, written in C, to the assertion.

Being written originally for C, App is function-oriented and therefore does not offer direct support for invariants, and certainly does not recognise the requirements of supporting Design by Contract across an inheritance hierarchy. Without the recognition of class invariants, the invariant must be repeated in the precondition and postcondition of every method of each class.

1.4.2.2 A++

Annotated C++ [xvi] was a research project (now sadly discarded) undertaken by Marshall Cline and Doug Lea. A++ was intended to provide designers with a means to express a variety of semantic constraints and properties of C++ classes. It extends the base type system to support several forms each of preconditions, postconditions, assertions and invariants, along with a richer set of "primitive" types and constructs that are useful for expressing such constraints.

At the same time, as a CASE tool, A++ attempts to perform formal verification on the annotations, statically verifying as far as possible the consistency of the implementation against the stated semantics[1]. Thus there would be the prospect of creating verified libraries of objects, making them attractive for reuse and therefore promoting one of the elusive objectives of object orientation.

Positioned as a front end to the normal C++ compiler, A++ is able to use annotations to make a number of significant improvements to the code, by eliminating run time consistency and exception tests that can be proven to be redundant. A++ therefore functions as an exception optimiser, reducing the exception testing overhead without sacrificing safety.

The tension between code efficiency and code safety, whereby the prohibitive cost of maintenance pushes design towards clarity and safety at the cost of speed, is reduced by A++, since it improves both code safety and efficiency.

1.4.2.2.1 Legality and Coherence

A++ makes an interesting distinction between legality and coherence for the state of an object. When data members are declared in a C++ class, it is implicit that objects of that class may enter states corresponding to all possible bitwise combinations of these data. Rarely though is it the case that the programmer actually intends this range of states to be used. Legality assertions identify that subset of the possible states that are not illogical, unreachable or undesirable. Objects must always be in a legal state.

Legality adds an extremely fine layer of granularity to the base type system. For example, in C++ the smallest subrange that is offered by the base type system is `unsigned char`, with values from 0 to 255. The legal assertions would be the place to declare that the variable *day_of_month*, which would have to be declared as a `char` or an `int` (unless an enumeration was used), has a lesser range.

A++ verifies that all constructors build legal objects, and then verifies – to the extent possible – that objects remain legal throughout the operation of each member function (whether public, protected or private).

The coherent states of an object of a class then correspond to a subset of the legal states and are analogous to self consistency. Coherence corresponds to the idea of an entity in Eiffel satisfying its invariant in all observable states. All public member functions may assume an object to be coherent when they are invoked, and are under obligation to ensure that if they disturb coherence during execution then they restore it before terminating. Thus there may be a coherence assertion relating the number of nodes in a linked list and the node count, which may be inconsistent during a node insertion operation (when perhaps the node has been inserted but the count has not yet been incremented) but not afterwards.

---

[1] C++ is amongst the most difficult languages to statically verify, mainly due to inconsistencies in the underlying type system of C such as pointer coercions, ambiguities between arrays and pointers, and aliasing.

Again, as best it can, A++ statically verifies that coherence is observed at the beginning and the end of the function (but not necessarily throughout). This is what is meant when saying an object remains coherent *across* each public member function.

1.4.2.2.2 Axioms

Whilst coherence (and hence legality) should be invisible to the user of the class (i.e. all objects should at all points of visibility never be anything other than coherent), there are numerous other conditions which are visible to the client. Placing objects in desirable states is the ultimate purpose of the member functions of a class, and these desirable states are described by the postconditions. Member functions can only produce these desirable states when objects are presented in certain initial states, described by the preconditions. Thus the behaviour of member functions is modelled as transitions between strict subsets of the set of coherent states. These subsets are highly visible to the clients of the class, and are sometimes have meaning for the client, such as "file is writable" or "stack is empty".

Unlike legality and coherence, which typically annotate a particular concrete representation, behavioural axioms use only implementation indepedent information (public member functions). This distinction is particularly apparent in the declaration of abstract base classes (ABCs). Typically an ABC includes mostly pure virtual member functions that define the protocol, but not the implementation, of a set of subclasses. C++ only guarantees that subclasses are syntactically conformant (and therefore does not necessarily ensure subtyping). The Stack below shows some typical axioms.

```
template <class T>
class Stack {
public:
  virtual void push(T)=0;
  virtual T pop()=0;
  virtual int length()=0;
  virtual void clear()=0;
  virtual int capacity()=0;
  virtual ~Stack() {}
  Stack() {}
  int full() { return length() == capacity(); }
  bool empty() { return !length(); }
axioms:
  [ require !full(); promise !empty(); T x ] push(x);
  [ require !empty(); promise !full() ] pop();
  [ promise return >= 0 ] length();
  [ promise empty() ] clear();
  [ promise return > 0 ] capacity();
  [ promise empty() ] Stack();
};
```

Writing ABCs provides many opportunities for reuse (Figure 1).

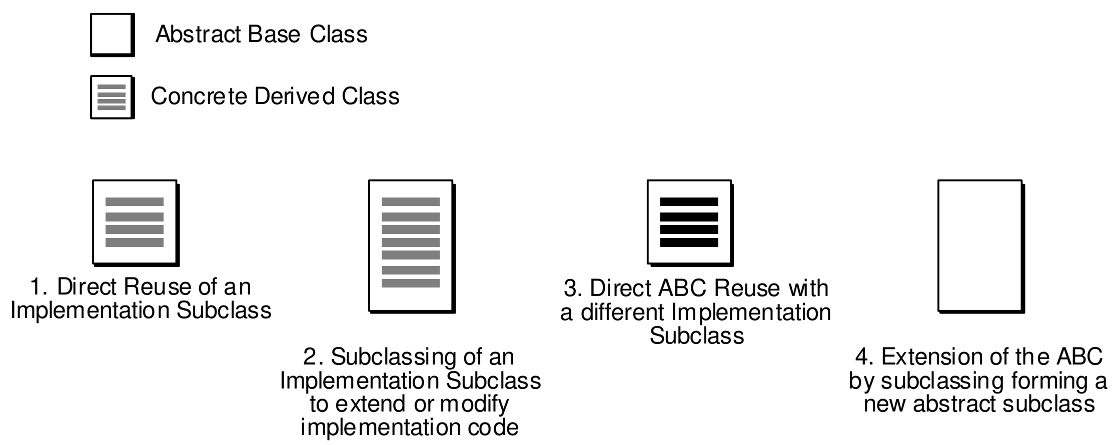

Figure 1. Possible reuse options for an ABC.

The strategy of separating specification from implementation supported and encouraged by A++ permits design reuse to occur practically independently of code reuse.

No mechanism is provided in A++ for inheriting behavioural constraints from privately inherited classes. Private inheritance is considered to express a HAS-A relationship, and to be equivalent to component inclusion.

In the example below, Vstack – an implemention of a Stack ABC using a Vector – HAS-A Vector (by inclusion) and IS-A Stack.

```
template <class T>
class VStack : public Stack<T> {
protected:
  Vector v;
  int sp;
legal: sp >= 0 && sp <= v.size();
public:
  VStack(int cap=10) : v(cap), sp(0) {}
  ~VStack() {}
  void push(T x) { v[sp++] = x; }
  T pop() { return v[--sp]; }
  void clear() { sp=0; }
  int length() { return sp; }
  int capacity() { return v.size(); }
};
```

As with Eiffel, member functions of derived classes are not allowed to require stronger preconditions or promise weaker postconditions than those of their base classes.

The annotation syntax of A++ has some distinctive features. Firstly, since A++ allows arbitrary statements and sequences of statements to be annotated, the annotations are put together, preceeding the statement, a convention later adopted in Sather.

```
  [ quantifers; require pre; promise post ] stmt
```

A novel axiom of A++ is **pure**; after execution of a **pure** function clients of a type cannot detect any change of state. For example,

```
template <class T>
class Stack {
// ..
axioms:
  [ T x; pure ] { push(x); pop(); }
// ..
};
```

This example also illustrates how close A++ came to the axiomatic expression of abstract data types.

A++ also takes into account friendship declarations. A class's legality constraint is automatically included as a precondition of every member function and friend. Furthermore, objects must also be coherent before execution of a public member or public friend.

1.4.2.3 exlC++

Although this language extension [xvii] bears a lot of similarity to much of the work discussed so far, the work of Porat & Fertig includes a number of new and interesting observations. Some of these are a result of taking the freedom to work outside of the existing language. Subsequent to the agreement of the ANSI C++ standard [xviii], this decision means that exlC++ is unlikely to make further contribution to the debate; however, some are still worth mentioning. Preconditions and postconditions are defined as part of the *declaration* of methods. For example,

```
typedef int Type;
const int BOS = -1;

class Stack
{
public:
  Stack(const int i)
    pre i>=0
    post {
      is_empty();
      size==i
    };
  ~Stack();
  int is_empty() const post post_result==(top==BOS);
  int is_full() const post post_result==(top==size-1);
  void push(const Type value)
    pre !is_full()
    post {
```

```
      !is_empty();
      array[top]==value;
      top==post_top(top)+1
    };
  void pop(const Type value)
    pre !is_empty()
    post {
      !is_full();
      post_result==array[post_old(top)];
      top==post_top(top)-1
    };
private:
  int top;
  int size;
  Type *array;

  invariant: {
    top >= BOS;
    top <= size-1;
    array != NULL;
  }
};
```

Defining preconditions and postconditions as part of the declaration of methods is a good way to highlight to fact that they are semantic properties and are independent of the implementation (although they may of course be expressed in a fashion that is dependent on the chosen implementation of the class).

The meaning of the semi-colons is equivalent to the && operator. The definition of an invariant can be split if desired. This may be useful if the class is large with many members and the invariant is composed of several conditions that are not logically related.

It is forbidden to attach preconditions and postconditions to static member functions. (The most common use of static member functions is to manipulate static member data. Static member data is shared by all objects of a given class. For example, a (non-static) member function may increment a private static data member with each call, to keep a tally of the number of calls. This tally would then be made accessible by providing a static member function which returns its value). The reason given for this limitation on static members is that assertions are considered to be conditions over an instance state, rather than just parameters alone. However, this does not appear to be a well thought-out justification. There seems no reason why a client would not want to state preconditions and postconditions relating to the use of functions that manipulate static data members as opposed to regular data members. The fact that the state of the static data in an object is shared by all objects of that class is not significant: it still contributes to the overall state of the system, and therefore it may be necessary to make assertions about the functions that operate on that data.

Semantically, assertions are restricted to Boolean-valued expressions. The intention is that they are side effect free. The compiler is required to check that assertions do not contain assignments and do not refer to non-`const` member functions.

Handling the properties of assertions across an inheritance hierarchy is given a thorough treatment, although there is no reference to the work of Liskov. Rules for dealing with preconditions and postconditions are proposed (and match those of the LSP) but there is a discussion of whether these rules are what is required in all cases. For example, the authors state that 'if a subclass provides additional attributes, it seems reasonable that preconditions within the subclass should be able to state conditions on these new attributes, in addition to the inherited assertions. For this requirement it is not necessary to have arbitrary access to the assertions of base classes, but preconditions (only) should be able to state an optional effective conjunct.' They do not mention that this proposal would allow derived classes that are not subtypes by any workable definition.

Abstract classes are also considered. It is often appropriate for class assertions to be associated with abstract classes. By providing preconditions and postconditions in the abstract class, not only is (compiler-enforcable) guidance supplied in a central place for subclass implementations, but subclasses are saved the effort of declaring the assertions themselves.

Class assertions can refer to all data members of the class. In particular, private data may be referred to. Clearly, when assertions are inherited by subclasses, the encapsulation of the superclass must not be broken. It is therefore the assertion *checks* that are inherited, and not the actual text of the assertions.

A distinction is therefore made between an *explicit* assertion that appears textually in a class and its implicit *effective* form. It is effective assertions that are actually checked; they contain references, in the form of sub-checks, both to the explicit assertion and to the (effective) assertions inherited from base classes, if any.

Another property of inheritance that is discussed is that of privacy. The authors propose that assertion checking should be independent of access. That is, if a class contains an inherited variable, then all the assertions that relate to that variable should be checked by that class, whether or not the functions of the class have direct access to the variable. Breaking encapsulation is avoided by the mechanism described in the previous paragraph.

If an assertion yields a false value, the run-time system will call a special function that defines the behaviour of the program in such a situation. There are three such special functions, _preFail, _postFail and invariantFail. These special functions are declared in a header file that must be included whenever the programmer uses compilation options that cause the compiler to generate run-time assertion checks. The definitions of these functions are either user-defined or by default provided in a library.

The default behaviour of coping with failure (as defined in the library) sends a representation of the following information on the violated assertion to the standard output stream.

- the type of assertion: preconditions, postcondition or invariant
- whether the failure occurred on entry or exit (invariant only)
- the name of the function member whose invocation led to the violation
- the names of the classes and files in which the relevant explicit assertions are declared
- an identification of the violated assertion items. Such identification is relevant for a precondition or postcondition if the assertion is a compound assertion. Identification of invariant assertion items is provided if the class invariant contains at least one compound assertion or if there are at least two invariant clauses that participate in the invariant check. If the assertion clause that failed is labeled, then its label serves to identify the individual clause.

1.4.3 Using Macros

Stroustrup says, "The first rule about macros is: Don't use them unless you have to. Almost every macro demonstrates a flaw in the programming language, the program or the programmer." It is clear that the problem in this case lies with the first of the three possibilities cited, and that the exploration of the use of macros to support Design by Contract in C++ is therefore justifiable.

1.4.3.1 Plessel

A rudimentary attempt to provide support for Design By Contract using macros is provided by Todd Plessel [xix]. A header file "Assertions.h" defines assertion macros for optionally verifying preconditions, postconditions and class invariants. The implementation uses the standard C `assert()` macro and requires the local class member function

```
virtual bool invariant() const
```

to be defined for each class. (The declaration of `invariant()` as `virtual` is questionable, since the property encoded by such a method is specific to the class and hence it could be argued that it should be bound statically).

Primary macros defined include:

INV	Assert class invariant.
PRE(c), PRE2(c1,c2), ... , PRE20(c1, ... c20)	Assert INV and pre-condition(s)
POST(c), POST2(c1,c2), ... , POST20(c1, ... c20)	Assert INV and post-condition(s)
CHECK(c), CHECK2(c1,c2), ... , CHECK20(c1, ... c20)	Assert arbitrary condition(s)

whilst support macros include:

OLD(variable)	For referring to previous values.
REMEMBER(type,variable)	Required if OLD() is used...
REMEMBER_F(type,function_name)	Same as above but for functions.

There are three levels of assertion checking and they are specified on the command line or in the Makefile:

(default)	Enables all assertion checking.
-DNO_ASSERTIONS	Disables all assertion checking.
-DPRECONDITIONS_ONLY	Enables PRE(c) checking only.

If any asserted condition evaluates to false at runtime then the program will display a message such as:

```
Assertion failed: (int)( index >= 0 ), file Test.c, line 44
```

An example of use is:

```
class Array10
{
private:
  int n;
  double a[10];
public:
  virtual bool invariant() const; // This is called by INV macro.
  bool insert( int i, double x ); // Insert x at a[ i ].
// ...
};

bool Array10::insert( int i, double x )
{
  PRE2( 0 <= i, i <= n )
  REMEMBER( int, n )
  // ...
  a[i] = x;
  // ...
  POST2( a[i] == x, n == OLD(n) + 1 ) // Also implies INV.
}
```

1.4.3.2 Nana

Gnu's Nana [xx] is a library that provides support for assertion checking and logging in a space and time efficient manner. The aim is to put common good practice into a library that can be reused. It was inspired by App. Assertion checking and logging code can be implemented using a debugger rather than as inline code with a large saving in code space. There is support for the C++ Standard Template Library (STL), in that there a special version of the macros involved which use the STL iteration protocol to process containers.

Nana is designed with the following concerns in mind:

(i) avoid the two executables problem. Nana is designed on the premise that normally people construct two versions of the program, one with checking code for testing and one without checking code for production use. The code space and time costs of having assertion checking and detailed logging code in a program can be high. With Nana, one version of the executable can be built for both testing and release since debugger-based checking has negligible space and time impact.

(ii) be configurable: the Nana library is designed to be reconfigured by the user according to their needs. For example it is possible to:

- modify the behaviour on assertion failure, e.g. to attempt a system restart rather than just shutting down.

- selectively enable and disable assertion checking and logging both at compile and run time.

- send the logging information off to various locations, e.g. user's terminal, a file for later checking, or to another process, (e.g. a plotting program or a program that verifies that the system is behaving correctly). A further option is to use a circular buffer in memory, which is very useful for production systems. The time cost of logging into memory in this way is not large and provides diagnostic information for occasions when the production system in the field has problems.

(iii) be time and space efficient. For example the GNU `'assert.h'` implementation uses 53 bytes for `'assert(i>=0)'` on a i386. The Nana version using the i386 `'stp'` instruction on assertion failure uses 10 bytes. This can be reduced to 0 or 1 byte by using debugger-based assertions, (although at the cost of execution speed).

(iv) provide support for formal methods. For example the `'isempty'` operation on a stack should leave the stack unchanged. To verify this in Nana the following could be used:

```
bool isempty(){ /* true iff stack is empty */
  DS($s = s); /* copy s into $s in the debugger */
  ...; /* code to do the operation */
  DI($s == s); /* verify that s hasn't been changed */
}
```

These `'$..'` variables are called convenience variables and are implemented by gdb. They have a global scope and are dynamically typed and initialised automatically to 0.

(v) provide support for Predicate Calculus. Nana provides some support for universal and existential quantification. For example to specify that the string v contains only lower case letters the following could be used:

```
I(A(char *p = v, *p != '\0', p++, islower(*p)));
```

These macros can be nested and used as normal boolean values in control constructs as well as assertions. Unfortunately they depend on the GNU CC statement value extensions and so are not portable.

Nana is C-based and as such suffers from the same shortcomings as App.

1.4.3.3 Welch and Strong

A simulation of the Eiffel Design by Contract mechanism put forward by Welch and Strong [xxi] defines macros which model the corresponding intrinsic language constructs in Eiffel.

The mechanism first adopts the increasingly standard idiom of redefining the

```
_assert(const char *, const char *, int)
```

macro. Normally, the `_assert(..)` macro tests a conditional expression, and if the result is false, a message is written to the default output device and the program is terminated using `abort()`.

The shortcomings of the existing version of `_assert(..)` are well known. The effect of a failed assertion is program termination. There is no opportunity to try to recover from the erroneous situation, and no opportunity to try to clean up the program state. In many cases this is simply unacceptable. Consequently assertions are very often compiled out of release versions of software. This can introduce an additional problem: the program logic can be altered by this process, since the conditional expression supplied to `assert(..)` may perform an action which then does not occur in the release version.

It is becoming standard practice to redefine `_assert(..)` so that the action taken when a test fails is that an exception is thrown. Redefining `_assert(..)` to throw an exception overcomes the worst of the problems of the existing `_assert(..)` macro. Detection of the error is separated from reporting and recovery by the exception mechanism. There is the opportunity to attempt to recover from the error and maintain program execution, and if not, to perform clean-up operations (e.g. closing open file handles, rolling back database transactions) before closing down.

The exception thrown is an instance of the class `Assertion`. This class is defined so as to be able to store the information passed to `_assert(..)` (the filename and line number where the violated assertion appears in the program), as well as the type of assertion that was violated.

```
class Assertion {
public:
// Type of assertion.
  enum Type
  {
```

```
    Require,
    Ensure,
    Check,
    Assert,
    Implies,
    NeverGetHere,
    ForAll,
    ThereExists
  };

  Assertion (Type assertionType, const char* reason, const char* file, int line);

  Assertion(const Assertion& other );
  virtual ~Assertion();

  Assertion& operator=(const Assertion& other );

  const char* getReason() const;
  const char* getFile() const;
  int  getLine()    const;
  Type getType()    const;

private:
  char*   theReason;
  char*   theFile;
  int     theLine;
  Assertion::Type theType;
};
```

The different types of assertion supported by the mechanism are

REQUIRE. States preconditions.

ENSURE. States postcondition.

CHECK. Checks external calls.

ASSERT. States a single property of the invariant.

IMPLIES. Tests a condition that must hold if another condition holds.

NEVER_GET_HERE. Verifies that a particular path is not encountered. Typically used to ensure that the "default" path of a select statement is not reached.

INVARIANT. States the conditions which must hold for a class object to be in a valid state.

CHECK_INVARIANT. Tests all properties of the INVARIANT.

BASE_INVARIANT. Tests the invariant of an inherited class. Can only be used within an INVARIANT block.

FOR_ALL. Applies a test to each item in a collection.

THERE_EXISTS. Applies a test which must hold for at least one item in a collection.

USES_OLD. Creates a copy, named "old", of the current object.

The definitions of the macros themselves are fairly standard. An example is the definition of CHECK.

```
// Check. Note that this assertion preserves the expression when disabled.
// This means that statements such as:
// CHECK((fi = fopen("file", "rb+" )) != NULL );
// are preserved when debugging is disabled.
#if defined ALL_ASSERTIONS || defined ASSERT_CHECK
#define CHECK(exp ) if(!(exp) ) { ASSERTION(Check, #exp ); ) else {;}
#else
#define CHECK(exp ) if(!(exp) ) {;} else {;}
#endif // defined ALL_ASSERTIONS || ASSERT_CHECK
```

Consider how the macro assertions appear in practice. Shown below is the code of a routine Stack::pop(int n), which removes n items from the top of the stack.

```
template <class T>
void Stack<T>::pop(int count)
{
  REQUIRE((count>0) && (count<=getCount()));
  USES_OLD(Stack<T>);
  theCount -= count;
  ENSURE(getCount()==old.getCount()-count);
  ENSURE(!isFull());
  CHECK_INVARIANT;
}
```

Welch and Strong's version of this routine included an additional assertion, but `REQUIRE(!isEmpty())` is redundant since it is implied by `REQUIRE(count>0)`.

The example illustrates how in this approach the preconditions and postconditions enclose the code of the routine body in a very similar fashion to the way that they do in Eiffel, and reflect the way that they would occur in a formal specification language such as VDM++.

However, there are also a number of drawbacks with the technique. Principally, there is no support for controlling assertion behaviour across an inheritance hierachy. There is no model of the Eiffel constructs **require else** and **ensure then**, and the conjunction of class invariants in a hierarchy has to be evaluated 'manually' by use of the BASE_INVARIANT macro. Secondly, the mechanism requires the user to explicitly state USES_OLD(..) when the post condition requires access to the value of the object state as it was prior to routine execution (and the user must also state the type of the object). Thirdly, CHECK_INVARIANT has to be explicitly called at the end of each routine – it is not an explicit part of the postcondition. At first sight this could be thought to be easy to add to the definition of the ENSURE macro; but many routines have multiple calls of ENSURE, and so the invariant would be checked many times. Finally, there is the familiar lack of guarantee that the testing of an assertion will not have an effect on program logic.

1.4.4 Summary of Design by Contract Attempts in C++

Table [] below summarises the significant points (good and bad) of the approaches to Design by Contract discussed in this chapter. This table will inform the discussion in the following chapter, where a new approach using genericity is considered. Considering the table, none of the approaches outlined (except using Eiffel) save the user from having to explicitly deal with handling the implications of inheritance for assertions. This seems to be a fundamental shortcoming, and was therefore taken to be the main problem to be addressed by the new approach.

	Eiffel	Percola-tion	App	A++	exlC++	Plessel	Nana	Welch & Strong
UNDESIRABLE								
Requires explicit coding to support the Liskov Substitutability Principle.		Yes	Yes			Yes	Yes	Yes
Clutters the class namespace with additonal names for precondition and postcondition routines.		Yes						
Can affect program execution (potentially has side-effects).	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Does not offer granularity of control over which assertions are executed.		Yes	Yes					
DESIRABLE								

Implementation readily available.			No	No	No			
Provides postconditions with access to the return value of the routine.		No				No	No	No
Provides postconditions with access to the state of the object at the commencement of the routine (the **old** value) without explicit coding.		No				No	No	No
Checking of the invariant is implicit in the postcondition.		No	No				No	No
Accommodates functions other than regular class member functions (static members, friends, global functions).	n/a	No		No				
Recognises and supports the distinction between legality and coherence.	No	No	No		No	No	No	No
Groups semantic information	No	No				No	No	No
Supports arbitrary numbers of parameters to member functions without requiring extension.		No				No	No	No
Takes advantage of non-strictness in the evaluation of preconditions and postconditions (executes preconditions and postconditions from the most derived class first).		No	No	No	No	No	No	No
Encapsulation is observed								

Consideration of the table permits a number of general points to be made:

- automatically providing postconditions with access to the return value of the routine (or the **old** value of a data item) is only possible with language extensions.
- it is not possible to provide full expressity in the assertion-checking mechanism without using functions which may have side-effects on program execution.
- automatically checking the invariant as an implicit part of the precondition and postcondition is only possible in the object-oriented schemes (i.e. those that were not originally C-based).

---

i Hoare, C.A.R.. The Emperor's Old Clothes. (1980 Turing Award lecture). *Communications of the ACM*, vol. 24, no. 2, February 1981, pp. 75-83.

ii Hoare, C.A.R. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, vol. 12, no. 10, October 1969, pp. 576-583.

iii Hoare, C.A.R. Proof of Correctness of Data Representations. *Acta Informatica*, vol. 1, 1972, pp. 271-281.

iv Floyd, Robert F. Assigning Meanings to Programs. *Proc. American Mathematical Society Symp. in Applied Mathematics*, vol. 19, 1967, pp. 19-31.

v Dijkstra, Edsger W. *A Discipline of Programming*. Prentice Hall, Englewood Cliffs, New Jersey, 1976.

vi Luckham, David C, et al. ANNA - A Language for Annotating Ada Programs. *Lecture Notes in Computer Science 260*, Springer-Verlag, 1987.

vii Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811-1841. November 1994.

viii W. Cook. A proposal for making Eiffel type-safe. Proceedings 3rd European Conference on Object-Oriented Programming, 57-70. 1989.

ix Betrand Meyer. Object-Oriented Software Construction, 2nd Edition. Prentice Hall, 1997.

x Bertrand Meyer. Applying Design by Contract. Computer (IEEE) vol 25 no. 10 p40-51 October 1992.

xi Omohundro, Stephen and David Stoutamire. Sather 1.1. http://www.icsi.berkeley.edu/~sather/Documentation/Specification/Sather-1.1/index.html. 1996.

xii Kramer, Reto. iContract - The Java Design By Contract Tool. http://www.promigos.ch/kramer. 1998.

xiii Robert V. Binder. Testing Object-Oriented Systems: Models, Patterns and Tools. Addison Wesley, 1999.

xiv Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Design Patterns for Object Oriented Software. Addison Wesley. 1994.

xv David Rosenblum. A Practical Approach to Programming with Assertions. IEEE Transactions on Software Engineering, Vol 21, No. 1, Jan 1995.

xvi Maurice Cline and Doug Lea. Using Annotated C++. Proceedings of C++ at Work. September 1990.

xvii Sara Porat & Paul Fertig. Class Assertions in C++. JOOP, May 1995.

xviii C++ International Standard ISO/IEC 14882, Section 14.7.3. ANSI 1998.

xix Todd Plessel. Design By Contract: A Missing Link In The Quest For Quality Software. http://www.elj.com/eiffel/dbc. 1998.

xx http://www.gnu.org/manual/nana/html_mono/nana.html

xxi David Welch & Scott Strong. An Exception-based Assertion Mechanism for C++. JOOP, July/August 1998.