A Modest Proposal: C++ Resyntaxed

Ben Werther & Damian Conway

Department of Computer Science, Monash University
Clayton, Victoria 3168, Australia
email: benw@netspace.net.au, damian@cs.monash.edu.au

Abstract

We describe an alternative syntactic binding for C++. This new binding includes a completely redesigned declaration/definition syntax for types, functions and objects, a simplified template syntax, and changes to several problematic operators and control structures. The resulting syntax is LALR(1) parsable and provides better consistency in the specification of similar constructs, better syntactic differentiation of dissimilar constructs, and greater overall readability of code.

1. Motivation

It is widely accepted that the syntax of C/C++, having been evolved over several decades by a large number of contributors, leaves much to be desired [1,2,3]. For example, the declaration syntax which C++ inherits from C (and extends) is so complicated that it is doubtful whether, without the assistance of a manual or source code example, one in ten C++ programmers could correctly declare a prototype for fundamental C++ allocation control function: **`set_new_handler`**[1]. We invite the reader who is familiar with C++ to attempt this exercise before continuing. The answer is given in Appendix A.

In *The Design and Evolution of C++,* Stroustrup observes that "within C++ there is a much smaller and cleaner language struggling to get out" [4] and foresees the development of "other interfaces" to C++. He cites Murray [5] and Koenig [6] who each demonstrated non-textual representations for C++ programs. Whilst such graphical representations of C++ offer considerable assistance in visualization and abstraction, they sacrifice some measure of the convenience, accessibility and portability of a purely ASCII representation.

We propose an alternative text-based syntactic binding (called SPECS[2]) for the existing semantics of the C++ language. The SPECS syntax departs substantially from the existing C++ syntax, particularly in the areas of declarations, definitions, templates, operators, and operator overloading.

Section 2 of this paper discusses some of the language design principles we employed in creating SPECS, whilst section 3 outlines the SPECS syntax and how it differs from the existing C++ syntax. Section 4 summarizes the remaining similarities between SPECS and C++. Note that this paper does not cover every feature of the SPECS syntax. For the complete specification of the SPECS language see [7].

2. SPECS Syntactic Design Principles

In designing the SPECS syntax we have been guided by four design principles: consistency, differentiation, readability, and formal grammatical simplicity.

The specification of types in C++ provides an excellent example of the *lack* of syntactic consistency that has resulted from the evolutionary development of the language. The following statements all define new types in C++:

```
class RXBase { virtual bool match(string) = 0; };
typedef struct { int x, y; int z; } Vector;
typedef int (*NumSrc)();
enum Result { reject, accept, defer };
union Tokens { int num; char* str; };
```

In designing the SPECS binding for C++ we have endeavoured to introduce better consistency into all syntactic forms, with particular attention to declaration syntax. In SPECS the above constructs are clearly indicated as creating new types with easily located names:

```
type RXBase : class { func match : abstract (string->bool); }
type Vector : class { [public] obj x, y: int; obj z : int; }
type NumSrc : ^(void->int);
type Result : enum { reject, accept, defer }
type Tokens : union { obj num : int; obj str : ^char; }
```

Interestingly, C++ also exhibits the opposite syntactic shortcoming – insufficient syntactic differentiation of dissimilar constructs. This is particularly evident in the declaration of functions and objects, where slight variations in component order may completely alter the meaning of a statement:

```
const Vector& (*vectorA)(int,Vector[]);
const Vector* (  vectorB)(int,Vector[]);
const Vector* (&vectorC)(int,Vector[]);
const Vector* &(vectorD)(int,Vector[]);
```

In SPECS the differences between these constructs are clearly indicated and their names are (once again) more easily ascertained:

---

1 which takes a single argument (a pointer to a function taking no arguments and returning **`void`**) and returns a similar function pointer (that is: it returns another pointer to a function taking no arguments and returning **`void`**).

2 "Significantly Prettier and Easier C++ Syntax"

```
obj  vectorA : ^((int, [] Vector) -> & const Vector);
func vectorB :   ((int, [] Vector) -> ^ const Vector);
obj  vectorC : &((int, [] Vector) -> ^ const Vector);
func vectorD :   ((int, [] Vector) -> & ^ const Vector);
```

Both these design goals help support a third – that of maximizing the overall readability of the language. Other notable contributions towards this goal have been:

- reordering declarations to emphasize important features and group together related information
- modification of the template declaration syntax to reduce excess verbosity and better localize relevant information;
- requiring that all iteration or selection statements are followed by brace-enclosed compound statements rather than single unbracketed statements;
- the introduction of the keyword **`common`** to specify "static" class members, thereby reducing the semantic overloading of **`static`**;
- the replacement of the old-style cast syntax with one that is similar to the new-style casts;
- the introduction of the **`abstract`** keyword to replace the "**`=0`**" syntax for pure virtual functions;
- the rebinding of the assignment operator to "**`:=`**" to better differentiate it from the equality test, which becomes "**`=`**";
- the rebinding of the "address of" operator to "**`@`**" so as to reduce the overloading of "**`&`**";
- the introduction of the **`defined_cast`** keyword to make operator conversion declarations more obvious;
- the addition of the keywords **`inherits`** and **`initially`** .

A final design principle for SPECS was grammatical simplicity. The language was designed to ensure that the new syntax was LALR(1) parsable, grammatically unambiguous and required no semantic feedback from parser to tokenizer. This constraint considerably simplifies the construction of a portable compiler for the language by allowing us to use the widely available parser construction tools *yacc* and *lex*. As mentioned above, this has also improved the readability of the language.

3. The SPECS syntax

Sections 3.1 to 3.5 summarize the principal differences between SPECS and (proto-)standard C++. Note that the bracketed symbolic names after each subheading indicate the corresponding section of the working paper for the draft ISO/ANSI C++ standard [8].

3.1. Declarations [dcl.dcl]

The area of greatest difference between SPECS and C++ is declaration syntax. The SPECS binding adopts a number of conventions for consistency across declaration types. Most significantly, all declarations begin with a keyword which identifies the declaration type. This keyword is then followed by the name of the entity being declared, where this is appropriate. Additionally, all declarations either end in a semicolon or a curly-brace enclosed block. This is also the case for C++ function definitions, however C++ type definitions ending in a curly-brace enclosed block require an additional trailing semicolon. SPECS is consistent across the entire syntax in neither requiring nor allowing such a trailing semicolon. Hence, in SPECS, a type ends either in a semicolon or a right brace, but never both.

3.1.1. Type IDs [dcl.name]

Specifying a type in C++ is made difficult by the fact that some of the components of a declaration (such as the pointer specifier) are prefix operators while others (such as the array specifier) are postfix. These declaration operators are also of varying precedence, necessitating careful bracketing to achieve the desired declaration. Furthermore, if the type ID is to apply to an identifier, this identifier ends up at somewhere between these operators, and is therefore obscured in even moderately complicated examples (see Appendix A for instance). The result is that the clarity of such declarations is greatly diminished.

Within SPECS, this problem is overcome by entirely redesigning the type ID mechanism in a manner similar to (but simpler than) that proposed by Anderson [1]. All declaration operators are prefix, right-associative, and are at the same precedence level. Any attached identifiers are separated from the type ID to make them visible. The intention is that the meaning of a type ID can be determined by simply reading it left-to-right, rather than by subtle parsing tricks better suited to a compiler rather than a programmer.

The following are simple C++ abstract declarators:

```
int              // integer
int  *           // pointer to integer
int  *[3]        // array of 3 pointers to integer
int  (*)[3]      // pointer to array of 3 integers
int  *()         // function having no parameters, returning pointer to integer
int  (*)(double) // pointer to function of double, returning an integer
```

The equivalent SPECS type IDs are:

```
int                  // integer
^ int                // pointer to integer
[3] ^ int            // array of 3 pointers to integer
^ [3] int            // pointer to array of 3 integers
(void -> ^int)       // function having no parameters, returning pointer to integer
^ (double -> int)    // pointer to function taking a double, returning an integer
```

The following table describes the operators and specifiers that can compose a declaration:

^ *typeID*	pointer to type *typeID*
ClassName*::^ *typeID	pointer to member (with type *typeID*) of class *ClassName*
& *typeID*	reference to type *typeID*
[] *typeID*	array of *typeIDs* with unspecified number of elements
[*constExpr*] *typeID*	array of *typeIDs* with *constExpr* elements
(*typeID*)	type *typeID*
(*paramList* -> *typeID*)	function taking *paramList* and returning *typeID*
const *typeID*	type constant *typeID*
volatile *typeID*	type volatile *typeID*

3.1.2. Type declarations and definitions [dcl.type]

In SPECS, all type declarations begin with the keyword **type**, followed by the type name. Four kinds of type declaration are allowed in SPECS: simple, enum, class, and union. The union declaration is analogous to that of the class and is not discussed here.

3.1.2.1. Simple type declaration

A simple type declaration in SPECS is equivalent to a C++ typedef. It associates an identifier with a type ID and has the syntax:

```
type identifier : typeID;
```

The following is a typical C++ typedef declaration:

```
typedef int* IntPtr;
```

The equivalent SPECS declarations is:

```
type IntPtr : ^ int;
```

3.1.2.2. Enum type declaration

An enum declaration is similar to its C++ equivalent, but in a format consistent with the other type declarations. It has the syntax:

```
type optional_enumName : enum { enumContents }
```

The following is a simple C++ enum declaration:

```
enum Colour { red=1, green, blue };
```

The equivalent SPECS declaration (declared *without* a trailing semicolon) is:

```
type Colour : enum { red:=1, green, blue }
```

3.1.2.3. Class type declaration

A class declaration in SPECS is semantically equivalent to a C++ class. It takes the place of both the C++ class and struct declarations, since the two constructs are isomorphic. A class definition has the form:

```
type className : class { memberDeclarations }
```

Section 3.2 describes the SPECS class definition mechanism in greater detail.

3.1.3. Object declarations and definitions [dcl.meaning]

An object declaration creates one or more variables. These variables can be of any type and need not just be instances of classes. All object declarations begin with the keyword **obj**. The general syntax is:

```
obj objName1, objName2, ... objNameN : typeID
```

where the type after the colon is one of the four types described in section 3.1.2.

The following are C++ declarations:

```
double *x, *y;
enum { green, gold } aColour;
```

The equivalent SPECS declarations are:

```
obj x, y : ^double;
obj aColour : enum { green, gold }
```

Objects can be initialized at construction time using either the "constructor" or "assignment" forms:

```
obj val1 := 1, val2(2) : int;
obj myInst(init1,init2) : MyClass::InnerType;
obj array := {1,2,3,4} : [4] int;
```

Different initialization syntaxes can be combined within one declaration statement, as in the first example. Note however that different types of variables cannot be instantiated in the same declaration. The infamous C++ example:

```
char* c1, c2, c3();
```

has no direct equivalent in SPECS. We consider this to be a feature.

Specifiers can be applied to objects at declaration time, and are listed directly after the colon in the declaration. Non-class-member objects can be declared **auto**, **register**, **static** and/or **extern**. Class members can be declared **common**, **mutable** and/or **bits(*constExpr*)** (to declare a bit field). For example:

```
obj localMax(1.0) : static register const double;
type MyClass : class { obj ourNextIndex : common int; }
```

Note that the **const** is not a specifier, but part of the actual type, and must therefore be placed after the specifier sequence.

Note too that SPECS reduces the overloading of the **static** keyword by introducing a new keyword, **common**, as a specifier for "static" class members. All other usages of **static** as a non-member storage specifier, as in C, are retained.

3.1.4. Function declarations and definitions [dcl.fct]

We note three main problems with the syntax of standard C++ function declarations:

- Function declarations are very similar in structure to variable declarations. In cases where the type of a variable involves a pointer to a function, the distinction can become very subtle, often involving only slight differences in bracketing.
- The name of a function has no uniform location within the declaration, and in more difficult examples can be embedded within levels of bracketing. Searching for a function declaration within a mass of code can be a challenge.
- The return type is placed before (and some distance away from) the parameter list, or may be entirely implicit. This makes it difficult to quickly ascertain the complete type of a function.

The SPECS syntax for function declaration tackles each of these problems. All function declarations begin with the keyword **func**, followed by the name of the function. The general syntax is:

```
func functionName : opt_specifiers ( parameterList -> returnType );
```

The parameter list is an optionally-bracketed, comma-separated list of parameters. The syntaxes for a parameter declaration are:

```
parameterName : paramType
parameterName := defaultValue : paramType
```

If a function takes no parameters, the parameter list must be declared **void**. The following are C++ function declarations:

```
fn1();
char* fn2(char* param1, int param2);
const T& fn3(int (*param1)[10]);
```

The equivalent SPECS declarations are:

```
func fn1 : (void -> int);
func fn2 : ((param1 : ^char, param2 : int) -> ^char);
func fn3 : ((param1 : ^ [10] int) -> & const T);
```

Specifiers can be applied to functions at declaration time. These are listed in sequence directly after the colon in the declaration. The specifier for a non-class-member object may be either **static** or **extern**. Class-member specifiers are: **inline**, **virtual**, **abstract**, **explicit** (constructors only) and/or **common** (replacing the C++ keyword **static** as in section 3.1.3).

The **abstract** keyword specifies that a member function is pure virtual. It is mutually exclusive with **virtual**, and can only be applied to declarations/definitions within a class:

```
type MyClass : class
{
[public]
      func printMe : abstract (& ostream -> void);
}
```

The equivalent C++ is:

```
class MyClass
{
public:
      virtual void printMe(ostream&) = 0;
};
```

3.1.5. Language declarations [dcl.asm, dcl.link]

A language declaration in a SPECS program specifies a (link to a) non-SPECS code fragment. It has the general structure:

```
lang "languageName" { declarationSeq }
```

where the set of allowed language names is implementation dependent, but contains at least **"asm"**, **"C"**, **"C++"** and **"SPECS".** The **lang** keyword provides a unification of the **asm** (assembly directive) and **extern** (external linkage) declarations of C++, whilst eliminating the semantic overloading of the latter.

Within an **"asm"** language declaration, assembly directives are either newline or semicolon terminated (or both):

```
lang "asm"
{
      mov eax, ebx; xchg ecx, edx
      shl eax, 7;        // This semi-colon not required
}
```

The equivalent C++ would be:

```
asm("mov eax, ebx"); asm("xchg ecx, edx");
asm("shl eax, 7");
```

The behavior of the **"C"** language declaration is identical to the corresponding extern (external linkage) declaration in C++. By extension, standard C++ syntax code is expected within a **"C++"** language declaration. This is useful when writing SPECS programs that include header files for one or more C++ standard libraries.

3.2. Class declarations and definitions [class]

A class declaration in SPECS is semantically equivalent to a C++ class. It takes the place of both the class and struct declarations, since the semantics of a struct can be achieved by the addition of a public access specifier at the start of a class type declaration. A class declaration has one of the following formats:

```
type className : class;
type className : class opt_inheritanceList { /* member declarations */ }
```

where the first form brings a class name into scope, and the second defines an actual class. The inheritance list has the form:

```
inherits BaseClass1, BaseClass2, ...etc
```

where the base class specifications may include one of the access specifiers: **public**, **protected** and **private**, and/or the inheritance specifier **virtual**, with the same semantics as in C++. Hence, the following SPECS class declaration:

```
type ListClass : class inherits public ListBase, private virtual ListImpl
{ /* member declarations */ }
```

is equivalent to the C++ declaration:

```
class ListClass : public ListBase, private virtual ListImpl
{ /* member declarations */ };
```

3.2.1. Member access control [class.access]

A member of a class can be declared public, protected or private (the default). Within a class, only one of these access specifiers will active at any time. To change the current specifier within a class declaration, a directive of the form **[*accessSpecifier*]** is used:

```
type PublicClass : class
{
      // members declared here are private
[public]
      // members declared here are public
}
```

3.2.2. Friends [class.friend]

In C++, a friend declaration is performed by placing the keyword **friend** in front of the declaration. We believe that this does not sufficiently differentiate between friend and member declarations. In SPECS, the keyword **[friend]** is used in exactly the same way as the access specifiers described in section 3.2.1. It declares all subsequent declarations to be friend declarations, until the end of the enclosing class or the next access specifier is reached:

```
type FriendlyClass : class
{
      // members declared here are private
[friend]
      // functions and types declared here are friends
}
```

3.2.3. Special member functions [special]

3.2.3.1. Constructor and destructor declarations

In C++ the names of a constructor and destructor of a class are derived from the class name. Because these names will differ from class to class, it can be difficult to identify these members at a first glance. To rectify this problem, in SPECS constructors and destructors are always called **ctor** and **dtor** respectively. The general syntax is:

```
func ctor : optional_specifiers ( paramList )
      optional_initializer_list
      { /* code */ }

func dtor : optional_specifiers ( void )
      { /* code */ }
```

Note that, as in C++, neither constructors nor destructors have a return type.

A constructor initializer list is introduced with the keyword **initially**:

```
func DerivedClass::ctor : (size : int, name : ^char)
      initially BaseClass::ctor(), mySize(size), myName(name)
      { /* code */ }
```

3.2.3.2. Operators and casts

The C++ operator overloading semantics are unchanged in SPECS but the syntax is modified to conform to the function declaration syntax described above and the renamed operators listed in section 3.4.1 below. In addition, two keywords, **pre** and **post**, have been introduced to simplify the overloading of the increment and decrement operators by replacing the awkward "dummy integer" syntax:

```
type MyInt  :  class
{
      obj  myVal  :  int;
[public]
      func  operator:=  :  (i:&  const  MyInt  ->  &  MyInt)
           {  if  (this@  !=  i@)  {  myVal  :=  i.myVal  }   return  this;  }
      func  operator  pre  ++  :  (void  ->  &  MyInt)
           {  myVal++;  return  this;  }
      func  operator  post  --  :  (void  ->  MyInt)
           {  obj  oldVal(this)  :  MyInt;  myVal--;  return  oldVal;  }
}
```

Note that in SPECS the identifier **this** within a member function acts as a reference, not as a constant pointer (as in C++).

In C++, a declaration of an operator conversion function has the name **operator *typeID*** and no return type. The SPECS declaration of an operator conversion function has the name **defined_cast**, and the target conversion type as the return type. For example, the equivalent of a C++ **operator int** member function in SPECS is declared:

```
type MyInt  :  class
{
[public]
      func  defined_cast  :  (void  ->  int)   {  return  myVal;  }
}
```

3.3. Template declarations and definitions [temp]

Although templates are a central feature of C++, the declaration syntax of C++ templates is not always easy to comprehend:

```
template<class  T1>  class  list
{
public:
      template<class  T2>  T1*  match(const  T2&);
};
```

It is even more cumbersome to define such a member of a template outside the template body :

```
template<class  T1>  template<class  T2>
T1  array<T1>::match(const  T2&)  {  /*  code  */  }
```

The problem stems from the fact that a template declaration has its template parameter list separated from the name that is being parameterized. SPECS alters this syntax so as to eliminate this separation and the excess verbosity of such declarations.

The first change is from simple angle brackets ("**<**" and "**>**") to composite brackets ("**<[**" and "**]>**"). Replacing single character brackets with two character brackets is not a change that was taken lightly, but it does solve three C++ problems and at the same time clearly delineates the parameterization of a template. The problems solved are:

- There is no longer a need to bracket template arguments containing the "greater than" operator, as in the C++ declaration **TemplateClass<(1>2)>**, since the SPECS version: **TemplateClass<[1>2]>** is unambiguous.
- It is no longer necessary to put a space between successive closing template brackets, (for example: **array<auto_ptr<X> >** in C++) . In SPECS, **array<[auto_ptr<[X]>]>** is unambiguous.
- The **template** keyword is not needed to disambiguate a member template function call, as in C++:

  ```
  x  :=  p->  template  alloc<200>();
  ```

  The SPECS equivalent is:

  ```
  x  :=  p^.alloc<[200]>();
  ```

3.3.1. Class templates

The general form of a template class is:

```
type className <[ templateParamList ]> : class opt_derivedList
{
      //  template  members...
}
```

The following C++ template examples:

```
template<class  Key,  class  Value>  class  Map  {  /*  members  */  };

template<class  T1>  template<class  T2>  class  Outer<T1>::Inner  {  /*  members  */  };
```

are written in SPECS as:

```
type  Map<[type  Key,  type  Value]>  :  class  {  /*  members  */  }

type  Outer<[type  T1]>::Inner<[type  T2]>  :  class  {  /*  members  */  }
```

Specializations of these templates in SPECS are similarly readable:

```
type Map<[int, ^char]> : class { /* specialized members */ }
type Outer<[type T1]>::inner<[int]> : class { /* specialized members */ }
```

3.3.2. Function templates [temp.fct]

The general form of a function template is:

```
func funcName <[ templateParamList ]> : opt_specifiers ( paramList -> returnType );
```

Usage is analogous to class templates (as in section 3.3.1).

3.4. Expressions [expr]

3.4.1. Operator changes

The C++ equality test (***val1* == *val2***) has been changed in SPECS to ***val1* = *val2***, as this binding for "**=**" is more consistent with widespread mathematical usage. The inequality operators ("**!=**", "**<**", "**<=**", "**>**", "**>=**") are unchanged.

As a consequence of the change in the equality operator, the C++ assignment operator (***ref* = *val***) becomes ***ref* := *val*** in SPECS[1]. As might be anticipated, the compound assignment operators become "**+:=**", "**-:=**" , "***:=**", etc.

The C++ unary prefix address-of operator (**&*ref***) becomes a postfix operator in SPECS: ***ref*@**. The unary prefix pointer dereference operator (***ptr*** in C++) also becomes a postfix operator: ***ptr*^**. As a result of this latter change to postfix notation, the C++ binary dereference-and-select-member operator (***ptr*->*member***) is no longer required in SPECS, as the syntax ***ptr*^.*member*** suffices. Likewise member selection through member pointers (**ptr->*memptr** and **ref.*memptr** in C++) become ***ptr*^.(*memptr*^)** and ***ref*.(*memptr*^)** respectively.

As a consequence of the use of "**^**" as the pointer dereference operator, the C++ binary bitwise exclusive-or operator (***bits1* ^ *bits2***) has been changed to ***bits1* ! *bits2*** in SPECS. The rationale for the choice of "**!**" is the analogy to "**!=**", the logical equivalent of xor.

The use of the operators **new** and **delete** is unchanged in SPECS, except in the case of the placement syntax, where a new keyword, **placement**, is reserved. The following C++ code:

```
ptr = new (myLocation) int[10];              // Placement new
::operator delete[](ptr,myLocation);         // Placement delete
```

becomes, in SPECS:

```
ptr := new [10] int placement(myLocation);
delete [] ptr placement(myLocation);
```

This both improves the readability of the expression and considerably simplifies the grammar. Note too that the array type on which **new** operates must conform to the declaration syntax described in section 3.1.1, and so the array size *precedes* the element type. This has the useful side-effect of making uniform the location of square brackets in calls to **operator new[]** and **operator delete[]**, which, unlike in C++, immediately follow the operator name in *all* cases in SPECS.

3.4.2. Change to the old-style casting syntax

New-style casts in C++ provide a clear indication that a cast is in progress, and the purpose of that cast. However the ISO/ANSI C++ standard committee has not deprecated the use of old-style casts. Hence, SPECS allows old-style casts, but requires a clearer syntax:

```
cast<[typeID]>(Expression)
```

3.5. Statements [stmt.stmt]

The **while**, **do-while** and **for** loops retain their C++ syntax in SPECS, except that a single unbracketed statement is no longer permitted as a loop body (that is, the body of a loop must be a block). Likewise, an **if** statement can only control a brace-enclosed block of zero or more statements, and the statement following an **else** must either be a (cascaded) **if** statement or a brace-enclosed block.

The **switch** statement in SPECS has been considerably altered from its C++ equivalent. It consists of zero or more specified cases and an optional default case, For example:

```
switch ( nextValue )
{
      case 1:        { cout << "Unity" << endl; }
      case 2,4,6,8: { cout << "Even" << endl; }
      default:       { cout << "Other" << endl; }
}
```

Each case consists of a list of one or more (integral) constant expressions, followed by a brace-enclosed block. If the condition matches any of the constant expressions belonging to a case, then the corresponding block of code will be executed. After the block executes control jumps to the end of the switch statement (*not* to the next case, as in C++). The **break** statement will also cause control to jump to

---

[1] Note that the arrow-like symbol "**<-**" was originally our preferred candidate for the assignment operator, but this binding creates significant problems because assignments can occur in logical tests, leading to ambiguities such as:

```
if (i<-10)  { cout << "big negative" << endl; }    // Assign 10 to i? Or compare i with -10?
```

the end of the switch statement. The **continue** statement can be used in a case block and causes control to jump immediately to the beginning of the next case block, thereby implementing a more general, but safer form of fall-through than the C++ default behaviour.

4. Commonalities with C++

By this stage the reader may feel that SPECS has little in common with its parent C++. In fact, the two languages are semantically isomorphic and significant portions of the two languages are syntactically identical. Features common to C++ and SPECS include:

- All inbuilt types are identically named and implemented. All literal values are specified in exactly the same way.
- The same set of operators (with the same precedences) are available for overloading. All binary operators maintain the same associativity in both languages.
- All scoping rules, temporary lifetimes, overloading constraints, function call resolution mechanisms, implicit conversions, access defaults and restrictions, and control structure semantics (except switch statement fall-through) are identical.
- The same preprocessor (*cpp*) and standard libraries are used for both languages.
- RTTI is identically implemented in both languages, and uses the same syntax.
- The exception handling mechanism and keywords are the same. The specification syntax is also identical except where syntactic differences in type or function specification preclude this.
- Namespaces have identical semantics and syntax in both languages.

Conclusion

In implementing a new text binding for the semantics of C++, we have enjoyed the unparalleled advantage of hindsight and the freedom to step beyond the restrictions of the evolutionary path of C++. Most significantly we have rejected the "failed experiment" of the C declaration notation, in favour of a more Pascal-like approach. We have also taken the opportunity to clean up other error-prone constructs and vestigial unpleasantnesses, such as fallthrough in a switch, the declaration of objects of (subtly) differing types in a single declaration, single statements as control structure bodies, **this** as a constant pointer (rather than a reference), the overuse of the "**&**" symbol and **static** and **extern** keywords, the "**=**"/"**==**" confusion, and the cryptic "**=0**" syntax for pure virtual functions.

The experience of syntactically redesigning a language of the complexity of C++ has been challenging and (at times) frustrating, and has left us with nothing but admiration for the designers of any real-world language, who must not only contend with all the issues and choices we have faced, but must also address the much harder task of simultaneously designing a consistent, powerful, and comprehensible semantics. It is our hope that the example of SPECS will encourage such language designers to recognize that language syntax is the physical interface between programmer and computer and, as such, demands just as much care and design as the language semantics, which is the programmer's logical interface to the machine.

References

[1] Anderson, B., *Type Syntax in the Language C: An Object Lesson in Syntactic Innovation*, in "Comparing and Assessing Programming Languages: Ada, C, and Pascal", Feuer & Gehani, eds., Prentice-Hall, 1984.

[2] Pohl, I. & Edelson, D., *A to Z: C Language Shortcomings, Computer Languages*, 13(2), pp. 51-64, Pergamon Press, 1988.

[3] Evans, A. *A Comparison of Programming Languages: Ada, Pascal and C*, in "Comparing and Assessing Programming Languages: Ada, C, and Pascal", Feuer & Gehani, eds., Prentice-Hall, 1984.

[4] Stroustrup, B., *The Design and Evolution of C++*, Section 9.4.4., Addison-Wesley, 1994.

[5] Murray, R., *A Statically Typed Abstract Representation for C++ Programs*, Proc. USENIX C++ Conference, Portland, Oregon, August 1992.

[6] Koenig, A., *Space Efficient Trees in C++*, Proc. USENIX C++ Conference, Portland, Oregon, August 1992.

[7] Werther, B., & Conway, D., *The Design and Implementation of SPECS: An alternative C++ syntax*, Computer Science Technical Report 96/256, Department of Computer Science, Monash University.

[8] *Working Paper for Draft Proposed International Standard for Information Systems – Programming Language C++,* ANSI Document X3J16/95–0087.

[9] Coplien, J., *Advanced C++: Programming Styles and Idioms*, Addison-Wesley, 1992.

Appendix A

The correct declaration of set_new_handler is:

```
void  (*set_new_handler(void  (*)(void)))(void);
```

This is so complicated that it is usually declared in two stages:

```
typedef  void  (*new_handler)(void);
new_handler  set_new_handler(new_handler);
```

The use of a typedef is not, in our opinion, a sufficient improvement.

The equivalent declarations in SPECS are considerably cleaner:

```
func  set_new_handler  :  (^(void->void)  ->  ^(void->void));
```

and:

```
type  new_handler  :  ^(void->void);
func  set_new_handler  :  (new_handler  ->  new_handler);
```

Appendix B

The following is a complete example of a simple Stack class in SPECS , provided so as to convey something of the "flavour" of the language. It is a line-for-line translation of the templated Stack class presented as Figures 7-1 and 7-2 in [9].

```
type Stack<[type T]> : class;

type Cell<[type T]> : class
{
[friend]
      type Stack<[T]> : class;
[private]
      obj next  : ^Cell;
      obj rep   : ^T;

      func ctor : (r:^T, c:^Cell<[T]>)
           initially rep(r), next(c)
           {}
}

type Stack<[type T]> : class
{
[public]
      func pop   : (void -> ^T);
      func top   : (void -> ^T)    { return rep^.rep; }
      func push  : (v:^T -> void)  { rep := new Cell<[T]>(v,rep); }
      func empty : (void -> int)   { return rep=0; }
      func ctor  : (void)          { rep := 0; }
[private]
      obj rep : ^Cell<[T]>;
}

func Stack<[type T]>::pop : (void -> ^T)
{
      obj ret(rep^.rep) : ^T;
      obj c(rep) : ^Cell<[T]>;
      rep := rep^.next;
      delete c;
      return ret;
}

func sort<[type S]> : ((elements: [] S, nelements: const int) -> void)
{
      obj flip:=0, sz:=nelements-1 : int;

      do
      {
              for (obj j:=0, flip:=0 : int; j<sz; j++)
              {
                     if (elements[j] < elements[j+1])
                     {
                          obj t:=elements[j+1] : S;
                          elements[j+1] := elements[j];
                          elements[j] := t
                          flip++;
                     }
              }
      } while (flip);
}
```