Persistence and Migration for C++ Objects*

Marc Shapiro, Philippe Gautron†, Laurence Mosseri

INRIA, B.P. 105, 78153 Le Chesnay Cédex, France, tel.: +33 (1) 39-63-53-25, telex: 697 033 F, e-mail: shapiro@sor.inria.fr

Abstract

We describe the support to object management of the distributed object-oriented operating system SOS. We discuss the integration of object migration and storage into C++ programs, a language not designed for that purpose. The necessary support is split between the compiler and a run-time object management system. Migration and storage preserve the type and structure of the objects, which may be user-defined and arbitrarily complex.

Our mechanisms are simple, generic, and require little programmer intervention. The key elements are dynamic classes, a generalized pointer type which allows to efficiently refer to an object, pre-requisite objects, and a mechanism for object re-initialization.

Keywords: *persistence, migration, dynamic linking, dynamic type-checking, C++, object-oriented operating system, SOS*

1 INTRODUCTION

We present some aspects of the object management support in the SOS operating system [Sha86a]. SOS is a research program to build a distributed operating system, where all interfaces and communication are based on objects.

In this paper, we explain the main features of SOS with respect to the migration and to the storage of complex objects. We will discuss the problems of programming in C++ when using object migration or accessing persistent objects. These problems

*This work was financed in part by the Commission of the European Communities under Esprit Contract 367 SOMIW.

†Current address: Labo Xerox-LITP, Université Pierre-et-Marie-Curie, 4 pl. Jussieu, 75232 Paris Cedex 05, France.

compelled us to slightly extend the language. Implications of the extension with respect to polymorphism and type-checking will be debated.

For migration and storage, SOS uses standard operating system techniques, such as paged virtual memory, untyped storage, and demand loading, instead of the more widespread textual flattening technique. A few simple and universal mechanisms, namely *pre-requisite* objects and *re-initialization* procedures, are the basis of our system.

Finally, a *generalized pointer*, the actual value of which is lazy-evaluated, allows unsophisticated programmers to deal efficiently with arbitrarily complex structures. We will show that in a standard object-oriented environment, with inheritance and parametric types, it is possible to implement a simple, automatic, efficient, fully type-checked object storage and migration facility.

2 SOS OBJECT MODEL

SOS supports arbitrary, medium sized (of the order of a hundred bytes or more) "SOS objects". For SOS, an object is simply some arbitrary collection of data, with some code attached to it.

Many elementary objects may execute together in a context or address space (similar to a Unix process with lightweight tasks); they may invoke each other via plain, type-checked, procedure calls.

Related elementary objects, executing in different contexts, may be grouped to form a single distributed object called a group. Elementary objets in a group have mutual communication privileges.

To gain access to a remote service, a client object must first acquire a *proxy* [Sha86b] for the service: a local interface object for the group implementing the service. The proxy is *migrated* at the time of need, into the client's context, e.g. at first access.

Object persistency is implemented (in first approximation) as migration to and from an Object Storage Server, which maintains a copy of the representation on disk. Thus, the problems of persistence are very similar to those of migration.

An object which must be known to SOS, such as a persistent object, will be of a class derived (i.e. inheriting) from class sosObject. This abstract class defines the interface to a few *virtual* member functions[1] which allow the SOS to invoke it. The programmer of a derived class X chooses to either inherit the sosObject code or override it with arbitrary actions.

When an object derived from sosObject is created, an inherited constructor for sosObject is first called, which implements all the mechanism for interfacing with SOS, e.g.

[1] *"Member function" is another word for "method". "Virtual" functions are also called "deferred" our "dynamically bound" in other contexts.*

allocating an entry for it in the object descriptor table. Thus, inheritance is enough to implement easy interfacing with the operating system.[2]

The object descriptor carries such information as a unique object ID (a 64 bit-number), group membership information, and references to *pre-requisite* objects, this object's required environment. For instance, an object cannot execute without its code; therefore its *code object* is necessarily one of its pre-requisites. We defer a fuller explanation of pre-requisites to section 4.2.

sosObject is not a universal root: classes which need not to be known to the OS at run-time don't have to derive from sosObject. In the sequel we will refer to such non-SOS objects as "plain objects". It is important to consider the case of plain objects for two reasons. First, sosObject management encurs a substantial overhead, and programmers should be allowed to use cheaper objects; this is used in particular for composite objects (section 6). Second, a Unix environment needs, and the C++ compiler itself generates, plain objects such as integers and character strings, which must remain accessible. Conversely, since plain objects cannot be migrated unless attached to an SOS object, they have much less functionality.

3 OBJECT MANAGEMENT IN STANDARD C++

We chose C++ because it is the most efficient object-oriented language we knew of; because of C compatibility; and because it is sufficiently low-level for systems work. C++ is typically used on a centralized system, such as Unix, with separate, transient address spaces, and no persistence.

In this section we will briefly present the main features of standard C++. The next section will present the extensions we felt compelled to introduce, in order to render painless the access to persistent/migratory objects.

3.1 Constructors and Destructors

In C++, the instantiation of an object runs a programmer-defined member function, the *constructor*.[3] Constructors are inherited; the constructor for a base class is called before the derived class's constructor. When it encounters a constructor, the compiler adds extra instructions, before or after the user code. These added instructions perform memory allocation, calls to the constructors of the base class and of member objects, and the assignment of the pointer to the "virtual table" (defined below). This extra generated code relieves users of some mechanical tasks, but deprives users of full mastery of execution. This is an obstacle to persistent/migratory objects.

[2]This remark is supported by the experience of others, e.g. [Det88,Dix86,Cam87].

[3]There is also a *destructor* called at object deletion. Everything we describe for constructors applies symmetrically to destructors.

3.2 Object Allocation and De-allocation

In C++ 2.0 [Str87a], a user-defined allocation and de-allocation strategy is possible on the free store by overloading operators **new** and **delete**, with a class granularity. No selective arguments other than the size can be specified in the user definition of these operators. This scheme lacks flexibility for migration and persistence, because an instance management granularity is required in this case.

Prior versions provided a different mechanism called "assign to this". The latter gives programmers finer control over memory management. Although it would be possible to base migration on it, it is inelegant, complex to use, and error-prone; therefore we rejected this solution in favor of the one described in section 4.

3.3 Virtual Member Functions

According to the taxonomy used in [Car85], C++ polymorphism is inclusion and ad-hoc polymorphism. Parametric polymorphism is currently contemplated [Str88].

Virtual member functions implement inclusion polymorphism. An object of a publicly derived class can be used in place of a base class instance. Virtual invocations involve a slight overhead at run-time because a virtual function call is indirect, via a table of procedure pointers (called the *virtual table*), accessed via a hidden pointer in the object's data. In C++, efficiency prevails: inclusion polymorphism is not the default.

All instances of a same class will share the same code and the same virtual table, i.e. there is only one implementation of a given class or type.

No mechanism allows users to access the virtual table. A similar conclusion to object allocation described above can apply: lack of flexibility in object management.

3.4 Conclusion

Standard C++ supports transient objects only, which are allocated during the lifetime of a process and de-allocated when that process finishes. It is impossible to add an object to a context without running a constructor, which necessarily allocates memory and initializes it. This is a serious impediment to both persistence and migration.

Furthermore, objects contain pointers (especially the pointer to the virtual table) which are meaningless if an object is migrated or read in from storage. Finally, a single implementation per class is supported, which poses problems for objects retrieved from storage.

4 C++ EXTENSIONS FOR SOS

The standard C++ mechanisms described above need to be extended to allow distributed object management, and object migration and persistence. The two main issues are: an input/output mechanism which escapes from the standard constructor/destructor mechanism, to permit *activation* of an existing (remote or stored) object without over-writing its value; and fixing the pointers in the data. Furthermore, when an object is activated at runtime, strong type-checking must be preserved. Finally, when an object is activated, the system must guarantee that the corresponding code and virtual table are available so that invocations are possible. In this section we explain the necessary extensions to C++.

The keyword dynamic has been added to the langage for two purposes:

- Declaration of a *dynamic class* enables dynamic binding of its code, in order to allow different implementations of a single C++ class (section 4.1), and
- *Dynamic instantiation* of a dynamic class performs object migration across address space boundaries or from disk, rather than local creation (section 4.2).

As explained in [Mor88], "*some form of dynamic binding is necessary in a persistent (object) system*". These extensions were designed to serve general purposes in the context of a distributed object environment, by increasing the flexibility in object management available to applications.

4.1 Dynamic Classes

A dynamic class is a class where all member functions, including constructors and destructors, are called indirectly via a *dynamic table*, an extension of the virtual table of section 3.1. It is destined to be dynamically loaded. This mechanism is more complete and more general than Stroustrup's proposal for incremental linking and loading [Str87b, p. 407-408]. Virtual member functions are allowed within a dynamic class.

A dynamic class declaration looks like this:

```
dynamic class X {
    // data and method declarations
};
```

All the references to the code of a dynamic class will be localized in its dynamic table, which simplifies dynamic linking. Dynamic linking is the task of relocating the code for the classes implementation, and constructing the appropriate dynamic table. This is detailed in section 4.4.

4.2 Dynamic Instantiation

Dynamic instantiation allows object migration across address space boundaries. A migration request is addressed to some server:

```
importRequest ir;
X* x = new dynamic("serverName") X(&ir);
```

The variable x receives a pointer to the imported instance. The following algorithm is used (see [Gau87,Sha88] for more details):

1. the argument ir is remotely passed to the server,
2. the server selects a "proxy" object to be migrated to the caller,
3. the proxy data and descriptor are migrated,
4. the proxy's pre-requisite objects (see section 4.3 below) are recursively migrated, if needed,
5. a local re-initialization occurs after the migration succeeded.

If the server is the Object Storage Server, the proxy is taken from disk. If it is some other server, it can select, either an object stored on disk, or an active object in its address space (possibly even itself). In either case, the proxy already exists somewhere at the time of the migration. It has thus been previously initialized by a constructor. Step 5 is a call to a "re-initialization constructor". A re-initialization constructor is different from others because its first argument is of type importRequest. Re-initialization constructors do not allocate memory; rather they use the data migrated by the system in step 3. In practice, most re-initialization constructors are no-ops, except for the code automatically inserted by the compiler.

4.3 Pre-requisites

The proxy's descriptor may contain a list of pre-requisite objects, i.e. its required environment. Obviously, one such pre-requisite[4] is the code.

In step 4 of the previous section, before the final re-initialization, the destination context is checked for those pre-requisites, and any missing ones are recursively imported.

[4]And the only kind currently implemented.

4.4 Code Objects

The code for a dynamic class (or, more exactly, the code for a particular implementation of a class) is encapsulated in an instance of the dynamic class code. A code instance contains all the necessary information to retrieve the compiled representation of the implementation and perform its dynamic link: class name, base class code reference, number of member functions (for constructing the dynamic table), and names of the corresponding binary files. The binary files are specially constructed so that the dynamic table, which is pre-allocated in its static data, is easy to find, via an extra field at the end of the string table.

A re-initialization constructor for a code object will perform dynamic type-checking, read into memory and relocate the binary files, and fill in the dynamic table. Then, a method allows to retrieve the address of the dynamic table.

4.5 Summary

The previous scenario of a dynamic instantiation for class X can now be completed, like this:

...

4. Recursive importation of the code object:
   - 4.1. The reference of the code object is sent to the storage server.
   - 4.2. The code object codeForX is selected.
   - 4.3. The code for X is copied in.
   - 4.4. The recursion stops here because codeForX has no further missing prerequisites.
   - 4.5. The re-initialization constructor of codeForX performs type-checking, dynamic linking, and binding of the dynamic table.

...

If codeForX was already present, either because it was linked statically, or because of a previous importation, this is skipped.

Later, another instance of class X may be imported. If it uses the same code object, these same steps will be skipped for it. If however it uses a different implementation (e.g. a different version, or a class derived from X) a different code object will be requested and imported. Its interface must be compatible, otherwise the type-checking of step 4.5 will fail.

4.6 C++ Polymorphism in SOS

The concept of code objects, described above, allows SOS-C++ to extend C++ inclusion polymorphism.

Different code implementations of a dynamic class can be dispatched on different instances, from the same server, or from different servers:

```
// -- class declaration
  dynamic class X {
        ...
  };

// -- instantiations
  importRequest ir1, ir2, ir3;
  X *x1, *x2, *x3;
  x1 = new dynamic ("serverName-1") X (&ir1);
  x2 = new dynamic ("serverName-1") X (&ir2);
  x3 = new dynamic ("serverName-2") X (&ir3);
```

Different dynamic tables can be assigned to x1, x2 and x3, and the corresponding code may possibly refer to a class publicly derived from X. They could react in a different way to the same invocations.

Note that it is the re-initialization constructor of the actual class which is called, as if it was a virtual. (C++ doesn't allow virtual constructors.) It is a real constructor, since the constructors for the base class and embedded objects are called as usual.

We conclude that the mechanisms presented here extend the C++ polymorphism rules in a compatible way. They perform type-checking for each implementation. Member function invocations encur no efficiency penalty.[5]

Type-safe migration requires some run-time type-checking. In SOS, its overhead is kept minimal by hashing the claimed type information in a 32-bit key provided by the compiler [Gau88]. Types are considered correct if the key supplied by the caller dynamic is equal to the key of the proxy's class, or of any of the compatible classes from which it inherits. This test conforms to the polymorphism rule. It is fast but not perfectly safe; this pragmatic approach has proved a realistic compromise.

5 STATE OF THE ART OF OBJECT INPUT AND OUTPUT

In many object-oriented systems I/O is performed by flattening out the representation, recursively calling encode/decode procedures [Her82] for each base class and

[5] *Of course, migration itself has a cost.*

each component object, with some mechanism to resolve forward references, and to break circularities. See for instance Stroustrup's proposal [Str87b, p. 409-410] or the description of the I/O facility for the OOPS C++ class library [Gor87]: encode does a recursive descent of the object, translating its internal, structured representation to a flat, ASCII-like representation. Such a system will output, along with an object, the transitive closure of the objects it refers to.

Encode/decode allows to translate between different representations. Since the encode/decode procedures are under programmer control, they can exploit the knowledge of the object's semantics. However this method requires programmer intervention, and is therefore error-prone. It is expensive, because of the translation overhead, because of the granularity of objects, because mapped I/O techniques cannot be used, and because the transitive closure technique is overkill. It does not preserve sharing.

The object migration mechanism of Emerald [Jul88] does not have these drawbacks. Suppose object X refers to object Y. X and Y may be in the same address space, or in different spaces. They may independently migrate to a new space. Emerald uses efficient pointers when possible, i.e. within the same space, and universal identification otherwise. X moves to a new space, simply by moving its data. Then the system updates the references it contains, appropriately for the destination space, using a compiler-generated "template", describing X. Emerald teaches us that I/O of an object can be greatly simplified, if the system has information about the references it contains.

6 PERSISTENT, COMPOSITE OBJECTS

In SOS, the Object Storage Service (OSS) handles the generic aspects of object persistency. It defines a minimal set of simple and generic tools to make storage of typed composite objects handled quasi-automatically by the system. Integrated with the migration mechanism, it gives the illusion of a single level store. The OSS uses a "lazy evaluation scheme" to resolve *intra-object references*. Indeed, it allows to benefit from standard operating system techniques, such as demand paged virtual memory, that reduce the complexity and minimize I/O. Data is transferred by segment[6]faulting, without encode/decode.

An SOS object may be arbitrarily complex, and contain references to other data. We manage differently references to **sosObjects** and to plain objects. A reference to an SOS object can only be resolved by migrating the object, as described above. In this section we describe how we handle the references to the plain-object components of an SOS object. When the pointed data is used, it must be present in the context. The references must be converted to contain valid addresses.

The data of an SOS object is organized into segments. Simple objects have a single "primary" segment. Composite objects have any number of additional "indirect"

[6]A segment is a kernel-level entity, a contiguous part of a virtual address space.

segments. The storage of such composite objects is handled by the system, provided pointers from one segment to another within the same object is a permPtr.

When a program needs to access an SOS object, it loads only its primary segment. This causes the invocation of its re-initialization constructor (see 4.2), and resetting the permPtr's in the primary segment (to be explained in section 7.1). Indirect segments are demand-loaded, only if actually accessed.

6.1 Data Input/Output

A permanent object is mapped to a disk file. As its segments are accessed, they are brought into memory. As the data is modified, the corresponding pages get marked by the virtual memory system, and are eventually paged back out to disk. When a permanent object is closed, either explicitly or because the address space terminates, it is unmapped, and all yet-unwritten pages are written out. For efficiency, all the segments of an object are coalesced into a single file.

When the data is written back to disk, no user code is run. This allows the system to write out arbritrary pages at arbitrary times.

(In our current prototype, built on top of Unix, we do not have the control of virtual memory. Therefore, we require the programmer to explicitly tell the system each time a segment is modified. When a permanent object is closed, each of its segments is written out in raw form, if it has been modified since the last output. In other words, hardware-supported demand paging is replaced by software-supported segmentation.)

7 INTRA-OBJECT REFERENCES

We wish to preserve the meaning of intra-object references across migration.

A plain pointer is the most efficient form of reference, but it must be translated at migration time. Since pointers are indistinguishible, at run-time, from ordinary data, they cannot be used for this purpose. An alternative is to use Object Identifiers (OIDs), which are universal and location-independent. This is not reasonable because first, the cost of dereferencing an OID is prohibitive; second, OIDs would impose an unnatural and error-prone programming style.

Therefore we define a new data type permPtr (for "permanent pointer"). A permPtr is a structure associating:

- a real C++ pointer to the datum;
- the (per-object) unique identifier of the segment in which the datum is included;
- a key within that segment, usually an offset;

The evaluation of permPtr's is explained in the next subsection.

7.1 Lazy Evaluation of References

The evaluation of a permPtr is to yield the address in the current context into the referenced objects. Once this address is known, it is stored in the "real pointer" field of the permPtr; subsequent evaluations simply looks up this value, and will have negligeable overhead.

When a segment is loaded, every "real pointer" field of each of the permPtr's it contains is reset to zero by the corresponding re-initialization constructor. The first time the permPtr is dereferenced, its pointer field is found to be null. The system looks up the unique identifier in its tables, and, if necessary, inputs the corresponding segment. Then, using the key field, the datum's new address is computed and stored. Finally, the re-initialization constructor of the pointed (plain) data is called. We use the re-initialization constructor also for plain objects to perform only class-specific data transformation, necessitated by the semantics of the data. For instance, the re-initialization constructor of permPtr objects has to reset its pointer field.

A permPtr to a base class can be set to a derived-class object. It is necessary in this case to store an indication to distinguish between multiple derived classes, in particular to call the correct re-initialization constructor. There is no dynamic type-checking or linking, as this was done previously when loading the enclosing SOS object.

7.2 Making the Evaluation of References Transparent

The generic type permPtr's replace ordinary pointers with very little effort, because C++ allows to define coercion operations. Whenever the compiler expects a pointer and sees a permPtr instead, the coercion operation is generated. The coercion operator calls the evaluation procedure described in the previous paragraph.

8 EXPERIENCE

A multi-media document manager, known as BFIR2, was implemented in co-operation with our Esprit partner Sarin, using the migration and storage facilities described above. In BFIR2, a document is a complex object, composed of a root node with descriptive information and (references to) a number of Chapter objects. These in turn contain (references to) Sections, which contain Paragraphs, which contain TextUnits. A TextUnit finally is a vector of bytes, which are interpreted either as ASCII text, or graphics, or a voice annotation, depending on what kind of Paragraph it is part of.

This is an example of logical structure of a document. BFIR2 can also deal with standardized layout formats.

The document class derives from the base class permObject. All other objects (Chapters, Sections, etc.) are plain objects. The BFIR2 document representation has served as a base for document exchange between various applications such as an editor or a

printer server. All the mechanism described above has been exercised by this application.

9 CONCLUSION

We have described a simple, fully type-checked, generic, object-oriented management support, suitable for object migration and storage service.

Although based on an untyped store, our approach is strongly typed, by taking advantage of the *pre-requisites* mechanism (sketched in section 4.3).

Our current implementation, in SOS V4 prototyped on top of Unix 4.2BSD (SunOS 3.4), has all the functionality described here.

Our Object Storage System is used by C++ programs. The integration with C++ is relatively painless. Most of the interfacing between applications and the system is done using standard C++ tools such as inheritance, constructors, and coercion methods. This suffices to make the creation of sosObject's and system-specific initializations, as well as the use of permPtr's, transparent to the casual user.

Parametric types are necessary to make permPtr work. These don't exist yet in C++, but are contemplated. In the meantime, they can be faked with macros.

Currently, persistency is a class attribute, in that an object is permanent only if its class derives from class permObject (which itself derives from sosObject). However, inheritance is not flexible enough for some uses. It should be possible to render persistent an individual instance of any class; and it should be possible at any time, not only at creation time. Furthermore, dynamic adaptation of storage policies (such as clustering) should be possible.

We are currently working on a new design based on pre-requisites, instead of inheritance. This should give the necessary flexibility.

Our major extension to C++/Unix is to allow objects to migrate, and to allow references to persistent/migratable objects. For this, we made a relatively small extension to the syntax of declarations. Once declared, such objects are used exactly in the same way as plain objects. Re-initialization constructors fall in naturally from the extension.

All of the mechanisms described here, implemented by a run-time library and a few support processes, are generic and have little to do with C++. We believe they can be used independently of the language, but we haven't yet experimented in that direction.

A Unix release of SOS-C++ is available on the USENIX C++ Distribution Tape. The implementation of SOS on top of Unix was useful for prototyping, but to go further, we need fine control of the way virtual memory is managed. Furthermore

the layering of SOS on top of Unix is very inefficient. We plan to use the Chorus V3 kernel [Roz88] in the near future.

ACKNOWLEDGEMENTS

Many thanks to all the SOR group who made this work possible, especially Sabine Habert for implementing object management, and for inventing prerequisites.

References

[Cam87] Roy Campbell, Vincent Russo, and Gary Johnston. The design of a multiprocessor operating system. In *Proceedings and additional papers, C++ workshop*, USENIX, Berkeley, CA (USA), November 1987.

[Car85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):472–522, December 1985.

[Det88] David Detlefs, Maurice Herlihy, and Jeannette Wing. Inheritance of synchronization and recovery properties in Avalon/C++. *Computer*, 21(12):57–69, December 1988.

[Dix86] G.N. Dixon and S.K. Shrivastava. *Exploiting Type Inheritance Facilities to Implement Recoverability in Object Based Systems.* Technical Report 223, University of Newcastle-upon-Tyne, Newcastle Upon Tyne, England, October 1986.

[Gau87] Philippe Gautron and Marc Shapiro. Two extensions to C++: a dynamic link editor, and inner data. In *Proc. C++ Workshop*, USENIX, Santa Fe NM (USA), November 1987.

[Gau88] Philippe Gautron. *Edition de liens dynamique, polymorphisme et réutilisabilité.* Note technique SOR-25, Projet SOR, INRIA, Rocquencourt (France), 1988.

[Gor87] Keith Gorlen. OOPS, a C++ object-oriented program support class library. In *USENIX C++ Workshop*, Santa Fe (New Mexico), November 1987.

[Her82] M. Herlihy and B. Liskov. A value transmission method for abstract data types. *ACM Transactions on Programming Languages and Systems*, 4(4):527–551, October 1982.

[Jul88] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[Mor88] R. Morrison, M.P. Atkinson, A.L. Brown, and A. Dearle. Bindings in persistent programming languages. *SIGPLAN notices*, 23(4):27–34, April 1988.

[Roz88] Marc Rozier, Vadim Abrossimov, François Armand, Ivan Boule, Frédéric Herrmann, Michel Gien, Marc Guillemont, Claude Kaiser, Pierre Léonard, Sylvain Langlois, and Willi Neuhauser. Chorus distributed operating systems. *Usenix Computing Systems Journal*, 1988.

[Sha86a] Marc Shapiro. SOS: a distributed object-oriented operating system. In *2nd ACM SIGOPS European Workshop, on "Making Distributed Systems Work"*, Amsterdam (the Netherlands), September 1986. (Position paper).

[Sha86b] Marc Shapiro. Structure and encapsulation in distributed systems: the Proxy Principle. In *Proc. 6th Intl. Conf. on Distributed Computing Systems*, pages 198–204, IEEE, Cambridge, Mass. (USA), May 1986.

[Sha88] Marc Shapiro. The design of a distributed object-oriented operating system for office applications. In *Proc. Esprit Technical Week 1988*, Brussels (Belgium), November 1988.

[Str87a] Bjarne Stroustrup. The evolution of C++ : 1985 to 1987. In *Proceedings and Additional Papers, C++ Workshop*, USENIX, Berkeley, CA (USA), November 1987.

[Str87b] Bjarne Stroustrup. Possible directions for C++. In *Proceedings and Additional Papers, C++ Workshop*, USENIX, Berkeley, CA (USA), November 1987.

[Str88] Bjarne Stroustrup. Parameterized types for c++. In *Proc. C++ Conference*, pages 1–18, USENIX, Berkeley, CA (USA), October 1988.