Chapter 6

Basic IDL-to-C++ Mapping

6.1 Chapter Overview

This chapter explains how IDL types are mapped to their corresponding C++ types by an IDL compiler. Sections 6.3 to 6.8 cover identifiers, modules, and simple IDL types. Section 6.9 covers memory management issues related to variable-length types, and Section 6.10 presents detailed examples of memory management for strings. Sections 6.11 and 6.12 discuss the mapping for wide strings and fixed-point types. The mapping for user-defined complex types is covered in Sections 6.13 to 6.18. Section 6.19 shows how smart pointers can eliminate the need to take care of memory management.

This chapter does not cover all of the mapping. Chapter 7 presents the client-side mapping for operations and exceptions, Chapter 9 details the server-side mapping, and Chapters 15 to 17 cover the dynamic aspects of IDL.

This chapter is long, and you probably won’t be able (or inclined) to absorb all of it by reading it from beginning to end. Instead, you may prefer to browse the sections that interest you and refer to the details later. The chapter is arranged so that it is suitable as a reference. All the material for a particular topic is presented together, so you should be able to find the answers to specific questions as they arise.
6.2 Introduction

The mapping from IDL to C++ must address a large number of requirements:

- The mapping should be intuitive and easy to use.
- It should preserve commonly used C++ idioms and “feel” like normal C++ as much as possible.
- It should be type-safe.
- It should be efficient in its use of memory and CPU cycles.
- It must work on architectures with segmented or hard (non-virtual) memory.
- It must be reentrant so that it can be used in threaded environments.
- The mapping must preserve location transparency; that is, the source code for client and server must look identical whether or not client and server are collocated (are in the same address space).

Some of these requirements conflict with others. For example, typically we cannot achieve ease of use and optimum efficiency at the same time, so we must make trade-offs. The C++ mapping adopted by the OMG deals with these compromises by choosing efficiency over convenience. The reason for this approach is twofold.

- It is possible to layer a slower but more convenient mapping on top of a faster but less convenient one, but we cannot layer a fast mapping on top of a slow one. Favoring a mapping that is fast but less convenient lets the OMG and ORB vendors add other options, such as code generation wizards, later.

- Increasingly, designers use IDL to describe in-process interfaces, which have the advantage of location transparency. Such interfaces let you build systems that implement different functional units in a single process and then let you later split that single process into multiple processes without breaking existing source code. The run-time efficiency of the mapping may be irrelevant for interprocess communication, but it matters for in-process communication.

These design choices mean that the C++ mapping is large and complex, but things are not as bad as they may seem. First, the mapping is consistent. For example, once you have understood the memory management of strings, you also know most of the rules for other variable-length types. Second, the mapping is type-safe; no casts are required, and many mistakes are caught at compile time. Third, the mapping is easy to memorize. Although some classes have a large number of member functions, you need call only a small number of them for typical use; some member functions exist to provide default conversions for parameter passing, and you need not ever call them explicitly.
Keep in mind that you should not try to read and understand the header files generated by the IDL compiler. The header files typically are full of incomprehensible macros, mapping implementation details, and cryptic workarounds for various compiler bugs. In other words, the header files are not meant for human consumption. It is far easier to look at the IDL instead. IDL and a knowledge of the C++ mapping rules are all you need to write high-quality code.

6.3 Mapping for Identifiers

IDL identifiers are preserved without change in the generated C++ code. For example, the IDL enumeration

```cpp
enum Color { red, green, blue };
```

groups to the C++ enumeration

```cpp
enum Color { red, green, blue };
```

The C++ mapping also preserves the scoping of IDL. If a scoped name such as `Outer::Inner` is valid in IDL, the generated C++ code defines the same name as `Outer::Inner`.

A problem arises if C++ keywords are used in an IDL definition. For example, the following IDL definition is legal:

```idl
enum class { if, this, while, else };
```

Clearly, this definition cannot be translated without mapping away from the C++ keywords. The C++ mapping specifies that IDL identifiers that are C++ keywords get a `_cxx_` prefix, so the preceding is translated as

```cpp
enum _cxx_class { _cxx_if, _cxx_this, _cxx_while, _cxx_else };
```

The resulting code is harder to read, so you should avoid using IDL identifiers that are C++ keywords.

It is also a good idea to avoid IDL identifiers containing a double underscore, such as

```idl
typedef long my__long;
```

The identifier `my__long` is legal and maps to C++ `my__long`. However, standard C++ reserves identifiers containing double underscores for the implementation, so, strictly speaking, `my__long` invades the compiler’s namespace. In practice, IDL identifiers containing double underscores are not likely to cause problems,
but you should be aware that the C++ mapping does not address this potential name clash.

### 6.4 Mapping for Modules

IDL modules are mapped to C++ namespaces. The contents of an IDL module appear inside the corresponding C++ namespace, so the scoping of an IDL definition is preserved at the C++ level. Here is an example:

```cpp
module Outer {
    // More definitions here...
    module Inner {
        // ...
    }
};
```

This maps to correspondingly nested namespaces in C++:

```cpp
namespace Outer {
    // More definitions here...
    namespace Inner {
        // ...
    }
}
```

A useful feature of namespaces is that they permit you to drop the name of the namespace by using a `using` directive. This technique eliminates the need to qualify all identifiers with the module name:

```cpp
using namespace Outer::Inner;
// No need to qualify everything
// with Outer::Inner from here on...
```

IDL modules can be reopened. A reopened module is mapped by reopening the corresponding C++ namespace:

```cpp
module M1 {
    // Some M1 definitions here...
};

module M2 {
    // M2 definitions here...
};
```
module M1 {  // Reopen M1
    // More M1 definitions here...
};

This maps to C++ as

namespace M1 {
    // Some M1 definitions here...
}

namespace M2 {
    // M2 definitions here...
}

namespace M1 {  // Reopen M1
    // More M1 definitions here...
}

Because not all C++ compilers have caught up with the ISO/IEC C++ Standard [9], namespaces are not universally available. For compilers not supporting namespaces, CORBA specifies an alternative that maps IDL modules to C++ classes instead of namespaces:

class Outer {
public:
    // More definitions here...
    class Inner {
        public:
            // ...
        }
    }
};

This alternative mapping is workable but has drawbacks.

• No using directive is available, so you must fully qualify names that are not in the current scope (or in one of its enclosing scopes).

• There is no sensible mapping of reopened modules onto classes. This means that IDL compilers will not permit you to reopen an IDL module if code generation is for a C++ compiler that does not support namespaces.

For the remainder of this book, we use the mapping to namespaces.
6.5 The CORBA Module

CORBA defines a number of standard IDL types and interfaces. To avoid polluting the global namespace, these definitions are provided inside the CORBA module. The CORBA module is mapped in the same way as any other module, so the ORB header files provide a CORBA namespace containing the corresponding C++ definitions.

We discuss the contents of the CORBA namespace incrementally throughout this book.

6.6 Mapping for Basic Types

IDL basic types are mapped as shown in Table 6.1. Except for string, each IDL type is mapped to a type definition in the CORBA namespace. The type definitions allow the mapping to maintain the size guarantees provided by IDL. To ensure that your code remains portable, always use the names defined in the CORBA namespace for IDL types (for example, use CORBA::Long instead of long to declare a variable). This will also help the transition of your code to 64-bit architectures (which may define CORBA::Long as int).

Note that IDL string is mapped directly to char * instead of a type definition. The reason is that when the OMG first produced the C++ mapping, it was felt that binary layout of data in memory had to be the same for both the C and the C++ mappings.\(^1\) This precludes mapping strings to something more convenient, such as a string class.

6.6.1 64-bit Integer and long double Types

The specification assumes that the underlying C++ implementation provides native support for (unsigned) long long and long double. If such support is not available, the mapping for these types is not specified. For that reason, you should avoid 64-bit integers and long double unless you are sure that they are supported as native C++ types on the platforms relevant to you.

\(^1\) In hindsight, imposing this restriction was probably a mistake because it forces the C++ mapping to be less type-safe and convenient than it could have been otherwise.
6.6 Mapping for Basic Types

6.6.2 Overloading on Basic Types

All the basic types are mapped so that they are distinguishable for the purposes of C++ overloading; the exceptions are char, boolean, octet, and wchar. This is because all three of the types char, boolean, and octet may map to the same C++ character type, and wchar may map to one of the C++ integer types or wchar_t. For example:

```c++
void foo(CORBA::Short param) { /*...*/ };
void foo(CORBA::Long param) { /*...*/ };
void foo(CORBA::Char param) { /*...*/ };
void foo(CORBA::Boolean param) { /*...*/ };  // May not compile
void foo(CORBA::Octet param) { /*...*/ };  // May not compile
void foo(CORBA::WChar param) { /*...*/ };  // May not compile
```

### Table 6.1. Mapping for basic types.

<table>
<thead>
<tr>
<th>IDL</th>
<th>C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>short</td>
<td>CORBA::Short</td>
</tr>
<tr>
<td>long</td>
<td>CORBA::Long</td>
</tr>
<tr>
<td>long long</td>
<td>CORBA::LongLong</td>
</tr>
<tr>
<td>unsigned short</td>
<td>CORBA::UShort</td>
</tr>
<tr>
<td>unsigned long</td>
<td>CORBA::ULong</td>
</tr>
<tr>
<td>unsigned long long</td>
<td>CORBA::ULongLong</td>
</tr>
<tr>
<td>float</td>
<td>CORBA::Float</td>
</tr>
<tr>
<td>double</td>
<td>CORBA::Double</td>
</tr>
<tr>
<td>long double</td>
<td>CORBA::LongDouble</td>
</tr>
<tr>
<td>char</td>
<td>CORBA::Char</td>
</tr>
<tr>
<td>wchar</td>
<td>CORBA::WChar</td>
</tr>
<tr>
<td>string</td>
<td>char *</td>
</tr>
<tr>
<td>wstring</td>
<td>CORBA::WChar *</td>
</tr>
<tr>
<td>boolean</td>
<td>CORBA::Boolean</td>
</tr>
<tr>
<td>octet</td>
<td>CORBA::Octet</td>
</tr>
<tr>
<td>any</td>
<td>CORBA::Any</td>
</tr>
</tbody>
</table>
The first three definitions of `foo` are guaranteed to work, but the final three definitions may not compile in some implementations. For example, an ORB could map IDL `char`, `boolean`, and `octet` to C++ `char` and map IDL `wchar` to C++ `short`. (In that case, the preceding definitions are ambiguous and will be rejected by the compiler.) To keep your code portable, do not overload functions solely on `Char`, `Boolean`, and `Octet`, and do not overload on `wchar` and an integer type even if it happens to work for your particular ORB.

### 6.6.3 Types Mappable to `char`

IDL `char`, `boolean`, and `octet` may map to signed, unsigned, or plain `char`. To keep your code portable, do not make assumptions in your code about whether these types are signed or unsigned.

### 6.6.4 Mapping for `wchar`

IDL `wchar` may map to a C++ integer type, such as `int`, or may map to C++ `wchar_t`. The mapping to integer types accommodates non-standard compilers, in which `wchar_t` is not a distinct type.

### 6.6.5 Boolean Mapping

On standard C++ compilers, IDL `boolean` may be mapped to C++ `bool`; the specification permits this but does not require it. If it is not mapped to C++ `bool`—for example, on classic C++ compilers—`CORBA::Boolean` maps to plain `char`, `signed char`, or `unsigned char`.

The C++ mapping does not require Boolean constants `TRUE` and `FALSE` (or `true` and `false`) to be provided (although `true` and `false` will work in a standard C++ environment). To keep your code portable, simply use the integer constants 1 and 0 as Boolean values; this works in both standard and classic environments.

### 6.6.6 String and Wide String Mapping

Strings are mapped to `char *`, and wide strings are mapped to `CORBA::WChar *`. This is true whether you use bounded or unbounded strings. If bounded strings are used, the mapping places the burden of enforcing the bound on the programmer. It is unspecified what should happen if the length of a
bounded string is exceeded at run time, so you must assume that the behavior is undefined.

The use of \texttt{new} and \texttt{delete} for dynamic allocation of strings is not portable. Instead, you must use helper functions in the \texttt{CORBA} namespace:

\begin{verbatim}
namespace CORBA {
    // ...
    static char * string_alloc(ULong len);
    static char * string_dup(const char *);
    static void string_free(char *);

    static WChar * wstring_alloc(ULong len);
    static WChar * wstring_dup(const WChar *);
    static void wstring_free(WChar *);
    // ...
}
\end{verbatim}

These functions handle dynamic memory for strings and wide strings. The C++ mapping requires that you use these helper functions to avoid replacing global \texttt{operator new[]} and \texttt{operator delete[]} and because non-uniform memory architectures may have special requirements. Under Windows, for example, memory allocated by a dynamic library must be deallocated by that same library. The string allocation functions ensure that the correct memory management activities can take place. For uniform memory models, such as in UNIX, \texttt{string_alloc} and \texttt{string_free} are usually implemented in terms of \texttt{new[]} and \texttt{delete[]}.

The \texttt{string_alloc} function allocates one more byte than requested by the \texttt{len} parameter, so the following code is correct:

\begin{verbatim}
char * p = CORBA::string_alloc(5);  // Allocates 6 bytes
strcpy(p, "Hello");              // OK, "Hello" fits
\end{verbatim}

The preceding code is more easily written using \texttt{string_dup}, which combines the allocation and copy:

\begin{verbatim}
char * p = CORBA::string_dup("Hello");
\end{verbatim}

Both \texttt{string_alloc} and \texttt{string_dup} return a null pointer if allocation fails. They do not throw a \texttt{bad_alloc} exception or a \texttt{CORBA} exception.

The \texttt{string_free} function must be used to free memory allocated with \texttt{string_alloc} or \texttt{string_dup}. Calling \texttt{string_free} for a null pointer is safe and does nothing.

Do not use \texttt{delete} or \texttt{delete[]} to deallocate memory allocated with \texttt{string_alloc} or \texttt{string_dup}. Similarly, do not use \texttt{string_free} to
deallocate memory allocated with `new` or `new[]`. Doing so results in undefined behavior.

The `wstring*` helper functions have the same semantics as the `string*` helper functions, but they operate on wide strings. As with `string_alloc`, `wstring_alloc` allocates an additional character to hold the zero terminating value.

### 6.7 Mapping for Constants

Global IDL constants map to file-scope C++ constants, and IDL constants nested inside an interface map to static class-scope C++ constants. For example:

```cpp
class NameList {
    static const CORBA::Long MAX_NAMES; // Classic or standard C++
    // OR:
    static const CORBA::Long MAX_NAMES = 20; // Standard C++
};
```

This mapping preserves the nesting of scopes used in the IDL, but it means that IDL constants that are nested inside interfaces are not C++ compile-time constants. In classic (non-standard) C++, initialization of static class members is illegal, so instead of generating the initial value into the header file, the IDL compiler generates an initialization statement into the stub file. Standard C++, on the other hand, permits initialization of constant class members in the class header for integral and enumeration types. Therefore, in a standard environment, you may find that constants defined inside an interface end up being initialized in the class header.

Normally, the point of initialization is irrelevant unless you use an IDL constant to dimension an array:
char * entry_array[MAX_ENTRIES];       // OK
char * names_array[NameList::MAX_NAMES]; // May not compile

You can easily get around this restriction by using dynamic allocation, which
works no matter how your IDL compiler maps constants:

char * entry_array[MAX_ENTRIES];       // OK
char ** names_array = new char *[NameList::MAX_NAMES]; // OK

String constants are mapped as a constant pointer to constant data:

const string MSG1 = "Hello";
const wstring MSG2 = L"World";

This maps to the following:

//
// If IDL MSG1 and MSG2 are at global scope:
//
const char * const MSG1 = "Hello";
const CORBA::WChar * const MSG2 = L"World";

//
// If IDL MSG1 and MSG2 are in an IDL interface "Messages":
//
class Messages {
public:
    static const char * const MSG1; // "Hello"
    static const CORBA::WChar * const MSG2; // L"World"
};

Note that if IDL constants are declared inside a module (instead of an interface),
their mapping depends on whether you are using a classic or a standard C++
compiler:

module MyConstants {
    const string GREETING = "Hello";
    const double PI = 3.14;
};

In classic C++, this maps to

class MyConstants {
public:
    static const char * const GREETING; // "Hello"
    static const CORBA::Double PI;      // 3.14
};
With a standard C++ compiler, the module maps to a namespace and the constants are in the generated header file:

```cpp
namespace MyConstants {
    const char * const GREETING = "Hello";
    const CORBA::Double PI = 3.14;
}
```

### 6.8 Mapping for Enumerated Types

IDL enumerated types map to C++ enumerations. The C++ definition appears at the same scope as the IDL definition. The enumeration is mapped to C++ unchanged except that a trailing dummy enumerator is added to force enumerators to be a 32-bit type:

```cpp
enum Color { red, green, blue, black, mauve, orange, _Color_dummy=0x80000000 // Force 32-bit size }
```

This appears in C++ as

```cpp
enum Color {
    red, green, blue, black, mauve, orange, 
    _Color_dummy=0x80000000 // Force 32-bit size
};
```

The mapping specification does not state what name is used for the dummy enumerator. The IDL compiler simply generates an identifier that will not clash with anything else in the same scope.

Note that this mapping guarantees that `red` will have the ordinal value 0, `green` will have the ordinal value 1, and so on. However, this guarantee applies only to the C++ mapping and not to all language mappings in general. This means that you cannot portably exchange the *ordinal values* of enumerators between clients and servers. However, you can portably exchange the enumerators themselves. To send the enumerator value `red` to a server, simply send `red` (and not zero). If `red` is represented by a different ordinal value in the target address space, the marshaling code translates it appropriately. (The mapping for enumerations is type-safe in C++, so you cannot make this mistake unless you use a cast. However, for other implementation languages, this may not be the case.)
IDL supports a number of variable-length types, such as strings and sequences. Variable-length types have special mapping requirements. Because the sizes of variable-length values are not known at compile time, they must be dynamically allocated at run time. This raises the issue of how dynamic memory is allocated and deallocated as well as your responsibilities as the programmer with respect to memory management.

The C++ mapping operates at two different levels. At the lower, or “raw,” level, you are responsible for all memory management activities. You can choose to code to this level, but the price is that you must remember exactly under what circumstances you need to allocate and deallocate dynamic memory. The lower level of the mapping also exposes you to differences in memory management rules for fixed- and variable-length structured types.

At the higher level, the C++ mapping makes life easier and safer by providing a set of smart pointer classes known as _var types. _var types relieve you of the burden of having to explicitly deallocate variable-length values and so make memory leaks less likely. These types also hide differences between fixed- and variable-length structured types, so you need not worry constantly about the different memory management rules that apply to them.

### 6.9.1 Motivation for _var Types

Programmers new to CORBA and the C++ mapping usually have difficulties coming to grips with _var types and understanding when and when not to use them. To clarify the motivation for _var types, let us consider a simple programming problem. The problem is not specific to CORBA; it applies to C and C++ in general. Here is the problem statement:

Write a C function that reads a string from an I/O device and returns that string to the caller. The length of the string is unlimited and cannot be determined in advance.

The problem statement captures a frequent programming problem, namely, how to read a variable-length value without advance knowledge of the total length of the value. There are several approaches to addressing the problem, and each has its own trade-offs.

**Approach 1: Static Memory**

Here is one approach to implementing the helper function:
const char *
get_string()
{
    static char buf[10000]; /* Big enough */
    /* Read string into buf... */
    return buf;
}

This approach has the advantage of simplicity, but it suffers from a number of serious drawbacks.

- The string to be returned may be longer than you expect. No matter what value you pick to dimension the buf array, it may be too small. If the actual string is too long, either you overrun the array and the code fails catastrophically, or you must arbitrarily truncate the string.
- For short strings, the function wastes memory because most of the buf array is not used.
- Each call to get_string overwrites the result of the previous call. If the caller wants to keep a previous string, it must make a copy of the previous result before calling the function a second time.
- The function is not reentrant. If multiple threads call get_string concurrently, the threads overwrite one another’s results.

**Approach 2: Static Pointer to Dynamic Memory**

Here is a second try at writing get_string:

const char *
get_string()
{
    static char * result = 0;
    static size_t rsize = 0;
    static const size_t size_of_block = 512;
    size_t rlen;

    rlen = 0;
    while (data_remains_to_be_read()) {
        /* read a block of data... */
        if (rsize - rlen < size_of_block) {
            rsize += size_of_block;
            result = realloc(result, rsize);
        }
        /* append block of data to result... */
6.9 Variable-Length Types and _var Types

```
rlen += size_of_block;
}
return result;
}
```

This approach uses a static pointer to dynamic memory, growing the buffer used to hold the data as necessary. Using dynamic memory gets rid of the arbitrary length limitation on the string but otherwise suffers the problems of the previous approach: each call still overwrites the result of the previous call, and the function is not reentrant. This version can also waste significant amounts of memory, because it permanently consumes memory proportional to the worst case (the longest string ever read).

**Approach 3: Caller-Allocated Memory**

In this approach, we make the caller responsible for providing the memory to hold the string:

```
size_t
get_string(char * result, size_t rsize)
{
    /* read at most rsize bytes into result... */
    return number_of_bytes_read;
}
```

This is the approach taken by the UNIX read system call. It solves most of the problems in that it is reentrant, does not overrun memory or arbitrarily truncate data, and is frugal with memory. (The amount of potentially wasted memory is under control of the caller.)

The disadvantage is that if the string is longer than the supplied buffer, the caller must keep calling until all the data has been read. (Repeated calls by multiple threads are reentrant if we assume that the data source is implicit in the calling thread.)

**Approach 4: Return Pointer to Dynamic Memory**

In this approach, get_string dynamically allocates a sufficiently large buffer to hold the result and returns a pointer to the buffer:

```
char *
get_string()
{
    char * result = 0;
    size_t rsize = 0;
    static const size_t size_of_block = 512;
```
while (data_remains_to_be_read) {
    /* read a block of data... */
    rsize += size_of_block;
    result = realloc(result, rsize);
    /* append block of data to result... */
}
return result;

This is almost identical to approach 2 (the difference is that get_string does not use static data). It neatly solves all the problems: the function is reentrant, does not impose arbitrary size limitations on the result, does not waste memory, and does not require multiple remote calls for long results (but dynamic allocation adds a little to the cost of collocated calls).

The main drawback of this approach is that it makes the caller responsible for deallocating the result:
/* ... */
{
    char * result;
    result = get_string();
    /* Use result... */
    free(result);

    /* ... */

    result = get_string();
    /* ... */
}
    /* Bad news, forgot to deallocate last result! */

Here, the caller returns from a block without deallocating the result returned by get_string. The memory occupied by the result can never be reclaimed. Repeated mistakes of this kind doom the caller to an inevitable death. Eventually, the caller runs out of memory and is aborted by the operating system, or, in an embedded system, the caller may lock up the machine.

6.9.2 Memory Management for Variable-Length Types

From the preceding discussion, it should be clear that approaches 1 and 2 are not suitable for the C++ mapping because they are not reentrant. Approach 3 is not an option, because the cost of repeated calls becomes prohibitive if caller and callee are on different machines.
This leaves approach 4, which is the approach taken by the C++ mapping for variable-length types. The C++ mapping makes the caller responsible for deallocating a variable-length result when it is no longer needed.

By definition, the following IDL types are considered variable-length:

- Strings and wide strings (whether bounded or unbounded)
- Object references
- Type any
- Sequences (whether bounded or unbounded)
- Structures and unions if they (recursively) contain variable-length members
- Arrays if they (recursively) contain variable-length elements

For example, an array of `double` is a fixed-length type, whereas an array of `string` is a variable-length type.

For each structured IDL type in a definition, the IDL compiler generates a pair of C++ types. For example, for an IDL union `foo`, the compiler generates two C++ classes: class `foo` and class `foo_var`. Class `foo` provides all the functionality required to use the union and corresponds to the lower mapping level. Class `foo_var` provides the higher mapping level by acting as a memory management wrapper around class `foo`. In particular, if class `foo` happens to represent an IDL variable-length type, class `foo_var` takes care of deallocating `foo` instances at the appropriate time.

The correspondence between IDL types and the lower and higher mapping levels is shown in Table 6.2.

<table>
<thead>
<tr>
<th>IDL Type</th>
<th>C++ Type</th>
<th>Wrapper C++ Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>string</code></td>
<td><code>char *</code></td>
<td><code>CORBA::String_var</code></td>
</tr>
<tr>
<td><code>any</code></td>
<td><code>CORBA::Any</code></td>
<td><code>CORBA::Any_var</code></td>
</tr>
<tr>
<td><code>interface foo</code></td>
<td><code>foo_ptr</code></td>
<td><code>class foo_var</code></td>
</tr>
<tr>
<td><code>struct foo</code></td>
<td><code>struct foo</code></td>
<td><code>class foo_var</code></td>
</tr>
<tr>
<td><code>union foo</code></td>
<td><code>class foo</code></td>
<td><code>class foo_var</code></td>
</tr>
<tr>
<td><code>typedef sequence&lt;X&gt; foo;</code></td>
<td><code>class foo</code></td>
<td><code>class foo_var</code></td>
</tr>
<tr>
<td><code>typedef X foo[10];</code></td>
<td><code>typedef X foo[10];</code></td>
<td><code>class foo_var</code></td>
</tr>
</tbody>
</table>
Note that structures, unions, and arrays can be fixed-length or variable-length. The IDL compiler generates a _var class even if the corresponding IDL type is fixed-length. For a fixed-length type, the corresponding _var class effectively does nothing. As you will see in Section 6.19, this class is useful for hiding the memory management differences between fixed-length and variable-length types. _var classes have similar semantics as the standard C++ auto_ptr template. However, the C++ mapping does not use auto_ptr (and other standard C++ types) because at the time the mapping was developed, many of the standard C++ types were not yet conceived.

We explore _var classes and their uses incrementally throughout the next few chapters. For now, we examine CORBA::String_var as an example of how _var classes help with dynamic memory management.

### 6.10 The String_var Wrapper Class

The class CORBA::String_var provides a memory management wrapper for char *, shown in Figure 6.1. The class stores a string pointer in a private variable and takes responsibility for managing the string’s memory. To make this more concrete, following is the class definition for String_var. We examine the purpose of each member function in turn. Once you understand how String_var works, you will need to learn little new for the remaining _var classes. The _var classes for structures, unions, and so on are very similar to String_var.

```cpp
class String_var {
public:
    String_var();
    String_var(char *);
    String_var(const char *);
    ~String_var();
    // etc...
private:
    char * s;
};
```

**Figure 6.1.** String_var wrapper class.
class String_var {
public:
    String_var();
    String_var(char * p);
    String_var(const char * p);
    String_var(const String_var & s);
    ~String_var();

    String_var & operator=(char * p);
    String_var & operator=(const char * p);
    String_var & operator=(const String_var & s);

    operator char *();
    operator const char *() const;
    operator char * &();

    char & operator[](ULong index);
    char operator[](ULong index) const;

    const char * in() const;
    char * & inout();
    char * & out();
    char * _retn();
};

String_var()
The default constructor initializes a String_var to contain a null pointer. If you use a default-constructed String_var value without initializing it first, you will likely suffer a fatal crash because the code ends up dereferencing a null pointer:

CORBA::String_var s;
cout << "s = \"" << s << \"\" << endl; // Core dump imminent!

String_var(char *)
This constructor initializes the String_var from the passed string. The String_var takes responsibility for the string: it assumes that the string was allocated with CORBA::string_alloc or CORBA::string_dup and calls CORBA::string_free when its destructor runs. The point is that you can initialize the String_var with a dynamically allocated string and forget about having to explicitly deallocate the string. The String_var takes care of deallocation when it goes out of scope. For example:
Basic IDL-to-C++ Mapping

```cpp
{ 
    CORBA::String_var s(CORBA::string_dup("Hello"));
    // ...
} // No memory leak here, ~String_var() calls string_free().

String_var(const char *)

If you construct a String_var using the const char * constructor, the String_var makes a deep copy of the string. When the String_var goes out of scope, it deallocates its copy of the string but leaves the original copy unaffected. For example:

```cpp
const char * message = "Hello";
// ...
```

```cpp
{ 
    CORBA::String_var s(message); // Makes a deep copy
    // ...
} // ~String_var() deallocates its own copy only.
```

```cpp
cout << message << endl; // OK
```

String_var(const String_var &)

The copy constructor also makes a deep copy. If you initialize one String_var from another String_var, modifications to one copy do not affect the other copy.

```cpp
~String_var
```

The destructor calls CORBA::string_free to deallocate the string held by the String_var.

String_var & operator=(char *)
String_var & operator=(const char *)
String_var & operator=(const String_var &)

The assignment operators follow the conventions of the constructors. The char * assignment operator assumes that the string was allocated with string_alloc or string_dup and takes ownership of the string.

The const char * assignment operator and the String_var assignment operator each make a deep copy.

Before accepting the new string, the assignment operators first deallocate the current string held by the target. For example:
6.10 The \texttt{String\_var} Wrapper Class

\begin{verbatim}
CORBA::String_var target;
target = CORBA::string_dup("Hello"); // target takes ownership

CORBA::String_var source;
source = CORBA::string_dup("World"); // source takes ownership

target = source; // Deallocates "Hello" and takes
// ownership of deep copy of "World".

operator char *()
operator const char *() const

These conversion operators permit you to pass a \texttt{String\_var} as a char * or
const char *. For example:

CORBA::String_var \texttt{s};
s = get_string(); // get_string() allocates with string_alloc(),
// s takes ownership

size_t len;
len = strlen(s); // const char * expected, OK

The main reason for the conversion operators is to let you transparently pass a
\texttt{String\_var} to IDL operations that expect an argument of type char * or
const char *. We discuss the details of parameter passing in Chapter 7.

operator char * &()

This conversion operator allows you to pass a string for modification to a function
using a signature such as

\begin{verbatim}
void update_string(char * &);
\end{verbatim}

Conversion to a \textit{reference} to the pointer (instead of just to the pointer) is necessary
so that the called function can increase the length of the string. A reference to the
pointer is passed because lengthening the string requires reallocation, and this in
turn means that the pointer value, and not just the bytes it points to, needs to
change.

char & operator[](ULONG)
char operator[](ULONG) const

The overloaded subscript operators permit you to use an index to get at the indi-
vidual characters of a \texttt{String\_var} as if it were an array. For example:

CORBA::String_var \texttt{s} = CORBA::string_dup("Hello");
cout << s[4] << endl; // Prints 'o'
\end{verbatim}
Strings are indexed as ordinary arrays are, starting at zero. For the "Hello" string, the expression s[5] is valid and returns the terminating NUL byte. Attempts to index beyond the NUL terminator result in undefined behavior.

6.10.1 Pitfalls of Using String_var

As you will see in Section 7.14.12, class String_var (and the other _var classes) exists mainly to deal with return values and out parameters for operation invocations. There are a number of situations in which String_var can be used inefficiently or inappropriately. Following are some of the pitfalls.

Initialization or Assignment from String Literals

String literals need special attention, at least if you are using classic (non-standard) C++; the type of a string literal is char * in classic C++ but is const char * in standard C++. If you are using a classic C++ compiler, the following code is guaranteed to crash sooner or later:

```cpp
CORBA::String_var s1("Hello"); // Looming disaster!
CORBA::String_var s2 = "Hello"; // Same problem!
```

Note that even though the second declaration looks like an assignment, it really is a declaration, and therefore both s1 and s2 are initialized by a constructor. The question is, which constructor?

In classic C++, the type of the string literal "Hello", when passed as an argument, is char *. The compiler therefore invokes the char * constructor, which takes ownership of the passed string. When s1 and s2 are destroyed, the destructor invokes string_free with an address in the initialized data segment. Of course, freeing non-heap memory results in undefined behavior and in many implementations causes a core dump.

The same problem arises if you assign a string literal to a String_var:

```cpp
CORBA::String_var s3;
s3 = "Hello"; // Calls operator=(char *), looming disaster!
```

Again, in classic C++, the type of "Hello" is char *(and not const char *), so the assignment is made by a call to String_var::operator=(char *). As with the char * constructor, this operator assigns ownership of the string to the String_var, and that will cause the destructor to attempt to free non-heap memory.
To work around this problem, either you can create a copy of the literal yourself and make the \texttt{String\_var} responsible for the copy, or you can force a deep copy by casting to \texttt{const char *}:

```cpp
// Note: For standard C++, you could use static\_cast\&<\texttt{const char *}> instead of the old-style (\texttt{const char *}) cast.

// Force deep copy
CORBA::String\_var s1((\texttt{const char *})"Hello");

// Explicit copy
CORBA::String\_var s2(CORBA::\texttt{string\_dup}("Hello"));

// Force deep copy
CORBA::String\_var s3 = (\texttt{const char *})"Hello";

// Explicit copy
CORBA::String\_var s4 = CORBA::\texttt{string\_dup}("Hello");

CORBA::String\_var s5;
s5 = (\texttt{const char *})"Hello"; // Force deep copy

CORBA::String\_var s6;
s6 = CORBA::\texttt{string\_dup}("Hello"); // Explicit copy

\texttt{const char *} p = "Hello"; // Make \texttt{const char *} pointer

CORBA::String\_var s7(p); // Make deep copy
CORBA::String\_var s8 = p; // ditto...
CORBA::String\_var s9;
s9 = p; // ditto...
```

The preceding code shows various ways of initializing and assigning string literals. In all cases, each \texttt{String\_var} variable ends up with its own separate copy of the literal, which can be deallocated safely by the destructor.

Wherever a cast to \texttt{const char *} is used, the constructor or assignment operator makes a deep copy. Wherever a call to \texttt{string\_dup} is used, a copy of the string literal is created explicitly, and the \texttt{String\_var} takes responsibility for deallocation of the copy.

Both approaches are correct, but as a matter of style we prefer a call to \texttt{string\_dup} instead of a cast. To a casual reader, casts indicate that something unusual is happening, whereas calling \texttt{string\_dup} emphasizes that an allocation is made.
The explicit copy style works correctly for both classic and standard C++, and we use that style throughout the remainder of this book. Of course, if you are working exclusively in a standard C++ environment, the following is safe:

```cpp
CORBA::String_var s = "Hello"; // OK for standard C++, deep copy
```

### Assignment of String_var to Pointers

If you assign a `String_var` variable to a `char *` or `const char *` variable, you need to remember that the assigned pointer will point at memory internal to the `String_var`. This means that you need to take care when using the pointer after such an assignment:

```cpp
CORBA::String_var s1 = CORBA::string_dup("Hello");
const char * p1 = s1; // Shallow assignment
char * p2;
{
    CORBA::String_var s2 = CORBA::string_dup("World");
    p2 = s2; // Shallow assignment
    s1 = s2; // Deallocate "Hello", deep copy "World"
} // Destructor deallocates s2 ("World")
```

```cpp
cout << p1 << endl; // Whoops, p1 points nowhere
cout << p2 << endl; // Whoops, p2 points nowhere
```

This code illustrates two common mistakes. Both of them arise from the fact that assignment from a `String_var` to a pointer is always shallow.

- The first pointer assignment (`p1 = s1`) makes `p1` point at memory still owned by `s1`. The assignment `s1 = s2` is a deep assignment, which deallocates the initial value of `s1` ("Hello"). The value of `p1` is not affected by this, so `p1` now points at deallocated memory.
- The second pointer assignment (`p2 = s2`) is also a shallow assignment, so `p2` points at memory owned by `s2`. When `s2` goes out of scope, its destructor deallocates the string, which leaves `p2` pointing at deallocated memory.

This does not mean that you should never assign a `String_var` to a pointer (in fact, such assignments are often useful). However, if you make such an assignment and want to use the pointer, you must ensure that the pointed-to string is not deallocated by assignment or destruction.
6.10 The String_var Wrapper Class

6.10.2 Passing Strings as Parameters for Read Access

Frequently, you will find yourself writing functions that accept strings as parameters for read access. Your program is also likely to have variables of both type char * and type String_var. It would be nice to have a single helper function that could deal with both types. Given the choice of char * and String_var, how should you declare the formal parameter type of such a function?

Here is how not to do it:

```cpp
void print_string(CORBA::String_var s) {
    cout << "String is \"" << s << "\" << endl;
}

int main() {
    CORBA::String_var msg1 = CORBA::string_dup("Hello");
    print_string(msg1); // Pass String_var
    return 0;
}
```

This code is correct but inefficient. The print_string function expects a parameter of type String_var. The parameter is passed by value, and that forces the compiler to create a temporary String_var instance that is passed to print_string. The result is that for every call to print_string, several function calls are actually made: a call to the copy constructor to create the temporary String_var instance that is passed to print_string, followed by a call to an overloaded ostream operator<< to print the string, followed by a call to the destructor to get rid of the temporary String_var again. The constructor calls string_dup (which calls strcpy), and the destructor calls string_free. The string_dup and string_free functions will probably call operator new[] and operator delete[], which in turn are often implemented in terms of malloc and free. This means that the preceding innocent-looking piece of code can actually result in as many as ten function calls for each call to print_string!

In most implementations, at least some of the function calls will be inlined, so the cost is not quite as dramatic as it may first seem. Still, we have observed massive slowdowns in large systems because of such innocent mistakes. Most of the cost arises from the hidden dynamic memory allocation. As shown in [11],
allocating and destroying a class instance on the heap is on average about 100 times as expensive as allocating and destroying the same instance on the stack.

Here is another problem with the print_string function:

```cpp
print_string("World"); // Call with char *, looming disaster!
```

This code compiles fine, and it prints exactly what you think it should. However, it will likely cause your program to dump core. This happens for the same reasons as discussed earlier: the type of the string literal is `char *` (at least in classic C++), and that eventually results in an attempt to deallocate non-heap memory in the destructor.

The key to writing `print_string` correctly is to pass a formal argument of type `const char *`:

```cpp
void
print_string(const char * s)
{
    cout << "String is \\
" << s << "\\n" << endl;
}
```

```cpp
int
main()
{
    CORBA::String_var msg1 = CORBA::string_dup("Hello");
    print_string(msg1); // Pass String_var, fine
    print_string("World"); // Pass as const char *, fine too
    return 0;
}
```

With this definition of `print_string`, things are well behaved. When the actual parameter is of type `String_var`, the compiler uses the `const char *` conversion operator to make the call. The conversion operator returns the private pointer inside the `String_var` and is typically inlined, and that keeps the cost of the call to a minimum.

Passing the string literal "World" to `print_string` does not create problems. The literal is simply passed as a `const char *` to the function.

No temporary is created in either case, and no calls to the memory allocator are necessary.

### 6.10.3 Passing Strings as Parameters for Update Access

To pass a string either as a `char *` or as a `String_var` to a function for update, a formal parameter of type `String_var &` will not work. If you pass a
char * where a String_var & is expected, the compiler creates a temporary. This results in construction of a String_var from a char * literal and eventually causes a core dump. To get it right, we must use a formal argument type of char * &:

```cpp
void update_string(char * & s)
{
    CORBA::string_free(s);
    s = CORBA::string_dup("New string");
}

int main()
{
    CORBA::String_var sv = CORBA::string_dup("Hello");
    update_string(sv);
    cout << sv << endl; // Works fine, prints "New string"

    char * p = CORBA::string_dup("Hello");
    update_string(p);
    cout << p << endl; // Fine too, prints "New string"
    CORBA::string_free(p);

    return 0;
}
```

A final warning: update_string assumes that the string it is passed was allocated with string_alloc or string_dup. This means that the following code is not portable:

```cpp
char * p = new char[sizeof("Hello")];
strcpy(p, "Hello");
update_string(p); // Bad news!
delete[] p;
```

This code causes a string allocated by new[] to be deallocated by string_free and causes a string allocated by string_dup to be deallocated by delete[], and that simply does not work on some platforms.

Calling update_string with an uninitialized pointer is also asking for trouble, because it results in passing a stray pointer to string_free, most likely with disastrous consequences. However, passing a variable initialized to null is safe; string_free does nothing when given a null pointer.
6.10.4 Problems with Implicit Type Conversions

Passing a String_var where a char * is expected relies on implicit type conversion. Some compilers do not correctly apply conversion operators, or they incorrectly complain about ambiguous calls. Rather than expect every C++ compiler to be perfect, the C++ mapping provides member functions that allow you to perform explicit conversions. These member functions are in, inout, out, and _retn (the names suggest the use for passing a parameter in the corresponding direction).

const char * in() const

You can call this conversion function if your compiler rejects an attempt to pass a String_var where a const char * is expected. For example:

```cpp
void print_string(const char * s) { /* ... */ } // As before

// ...

CORBA::String_var sv(CORBA::string_dup("Hello"));
print_string(sv); // Assume compiler bug prevents this
print_string(sv.in()); // Explicit call avoids compiler bug
```

The in member function returns the private pointer held by the String_var wrapper as a const char *. You could achieve the same thing by using a cast:

```cpp
print_string((const char *)sv);
```

This code explicitly invokes operator const char * on the String_var. However, using the in member function is safer than a “sledgehammer” cast that bypasses all type checking. Similar arguments apply to using the inout and out member functions in preference to a cast.

char * & inout()

You can call the inout member function if your compiler refuses to accept a String_var where a char * & is expected. For example:

```cpp
void update_string(char * & s) { /* ... */ } // As before

// ...

CORBA::String_var sv;
update_string(sv); // Assume compiler bug prevents this
update_string(sv.inout()); // Explicit call avoids compiler bug
```
The **inout** member function returns a reference to the pointer held by the **String_var** wrapper so that it can be changed (for example, by reallocation).

```cpp
char * & out()
```

This conversion operator allows you to pass a **String_var** as an **output** parameter where a `char * &` is expected. The **out** member function differs from the **inout** member function in that it deallocates the string before returning a reference to a null pointer. To see why this is necessary, consider the following helper function:

```cpp
void
read_string(char * & s) // s is an out parameter
{
    // Read a line of text from a file...
    s = CORBA::string_dup(line_of_text);
}
```

The caller can use `read_string` as follows without causing a memory leak:

```cpp
CORBA::String_var line;
read_string(line.out()); // Skip first line
read_string(line.out()); // Read second line - no memory leak
cout << line << endl;    // Print second line
```

Calling the **out** member function does two things: it first deallocates whatever string is currently held by the **String_var**, and then it returns a reference to a null pointer. This behavior allows the caller to call `read_string` twice in a row without creating a memory leak. At the same time, `read_string` need not (but can) deallocate the string before allocating a new value. (If it deallocates the string, no harm is done because deallocation of a null pointer is safe.)

### 6.10.5 Yielding Ownership of a String

The **_retn** member function returns the pointer held by a **String_var** and also yields ownership of the string. This behavior is useful if a function must return a dynamically allocated string and also must worry about error conditions. For example, consider a **get_line** helper function that reads a line of text from a database. The caller uses the function this way:

```cpp
for (int i = 0; i < num_lines; i++) {
    CORBA::String_var line = get_line();
    cout << line << endl;
} // Destructor of line deallocates string
```
Consider how this works. The get_line function dynamically allocates the returned string and makes the caller responsible for deallocation. The caller responds by catching the return value in the String_var variable line. This makes line responsible for deallocating each returned line in its destructor. Because line is declared inside the body of the loop, it is created and destroyed once per iteration, and the memory allocated to each line is deallocated immediately after each line is printed.

Following is an outline of the get_line function. The important point is that get_line may raise an exception after it has allocated the string:

```cpp
char *
get_line()
{
    // Open database connection and read string into buffer...

    // Allocate string
    CORBA::String_var s = CORBA::string_dup(buffer);

    // Close database connection
    if (db.close() == ERROR) {
        // Whoops, a serious problem here
        throw DB_CloseException();
    }

    // Everything worked fine, return string
    return s._retn();
}
```

The trick here is that the variable s is a String_var. If an exception is thrown sometime after memory is allocated to s, there is no need to worry about memory leaks; the compiler takes care of invoking the destructor of s as it unwinds the stack to propagate the exception.

In the normal case, in which no error is encountered, get_line must return the string and make the caller responsible for it. This means that get_line cannot simply return s (even though it would compile), because then the string would be incorrectly deallocated twice: once by the destructor of s, and a second time by the caller.

The final statement in get_line could be the following instead:

```cpp
return CORBA::string_dup(s);
```

This code is correct but makes an unnecessary and expensive copy of the string. By invoking the _retn member function instead, get_line transfers responsi-
bility for deallocating s to the caller. This technique leaves the string in place and avoids the cost of making a copy.

6.10.6 Stream Operators

The C++ mapping provides overloaded String_var insertion and extraction operators for C++ iostreams:

```c++
CORBA::String_var s = ...;
cout << "String is " << (s != 0 ? s : "") << "\n" << endl;
cin >> s;
cout << "String is now " << (s != 0 ? s : "") << "\n" << endl;
```

Overloaded operators are provided for istream and ostream, so they can also be used with string (strstream) and file (fstream) classes.

6.11 Mapping for Wide Strings

The mapping for wide strings is almost identical to that for strings. Wide strings are allocated and deallocated with the functions wstring_alloc, wstring_dup, and wstring_free (see page 147). The mapping also provides a WString_var class (in the CORBA namespace) that behaves like a String_var but operates on wide strings.

6.12 Mapping for Fixed-Point Types

C++ does not have built-in fixed-point types, so C++ support for fixed-point types and arithmetic is provided by a class and a number of overloaded operator functions:

```c++
namespace CORBA {
    // ...
    class Fixed {
    public:
        Fixed(int val = 0);
        Fixed(unsigned);
        Fixed(Long);
        Fixed(LongLong);
        Fixed(UInt);
        Fixed(Double);
    ```
Fixed(LongDouble);
Fixed(const char *);

Fixed(const Fixed &);
~Fixed();

operator LongLong() const;
operator LongDouble() const;
Fixed round(UShort scale) const;
Fixed truncate(UShort scale) const;

Fixed & operator=(const Fixed &);
Fixed & operator+=(const Fixed &);
Fixed & operator-=(const Fixed &);
Fixed & operator*=(const Fixed &);
Fixed & operator/==(const Fixed &);

Fixed operator++();
Fixed operator++(int);
Fixed operator--();
Fixed operator--(int);
Fixed operator+() const;
Fixed operator-() const;
int operator!() const;

UShort fixed_digits() const;
UShort fixed_scale() const;

istream & operator>>(istream &, Fixed &);
ostream & operator<<(ostream &, const Fixed &);

Fixed operator+(const Fixed &, const Fixed &);
Fixed operator-(const Fixed &, const Fixed &);
Fixed operator*(const Fixed &, const Fixed &);
Fixed operator/(const Fixed &, const Fixed &);

int operator<(const Fixed &, const Fixed &);
int operator>(const Fixed &, const Fixed &);
int operator<=(const Fixed &, const Fixed &);
int operator>=(const Fixed &, const Fixed &);
int operator==(const Fixed &, const Fixed &);
int operator!=(const Fixed &, const Fixed &);

// ...
}
This mapping enables you to use fixed-point quantities in C++ and to perform computations on them. Note that a single generic Fixed class is used, so the IDL compile-time digits and scale for fixed-point types become run-time values in C++.

6.12.1 Constructors

The Fixed class provides a number of constructors that permit construction from integer and floating-point types.

The default constructor initializes the value of a Fixed to zero and internally sets the digits to 1 and the scale to 0—that is, the value has the type fixed<1,0>.

Constructing a Fixed value from an integral value sets the digits to the smallest value that can hold all the value’s digits and sets the scale to zero:

Fixed f = 999; // As if IDL type fixed<3,0>

Constructing a Fixed value from a floating-point value sets the digits to the smallest value that can represent the floating-point value. The scale is set to preserve as much of the fractional part of the floating-point value as possible, truncating at the relevant digit. Here are a few examples:

Fixed f1 = 1000.0; // As if IDL type fixed<4,0>
Fixed f2 = 1000.05; // As if IDL type fixed<6,2>
Fixed f3 = 0.1; // Typically as if IDL type fixed<18,17>
Fixed f4 = 1E30; // As if IDL type fixed<31,0>
Fixed f5 = 1E29 + 0.89; // As if IDL type fixed<31,1>,
                    // value is 1E29 + 0.8

Note that initialization from floating-point values can result in surprising digits and scale because of the vagaries of binary floating-point representation. For example, the value 0.1 results in an actual value of 0.10000000000000001 in many implementations. Also note that even though the value 1E29 + 0.89 is treated as 1E29 + 0.8 for the purpose of truncation, it is unlikely that your C++ compiler will be able to represent floating-point numbers with the required precision. For example, on many implementations, the Fixed value will be initialized to 9999999999999999100000000000000000000 instead.

Initialization with a value that has more than 31 integral digits throws a DATA_CONVERSION exception (see Section 7.15 for details on exception handling):

Fixed f = 1E32; // Throws DATA_CONVERSION
Constructing a `Fixed` value from a string follows the rules for IDL fixed-point constants (see Section 4.21.4). Leading and trailing zeros are ignored, and a trailing “D” or “d” is optional:

```cpp
Fixed f1 = "1.3";    // As if fixed<2,1>
Fixed f2 = "01.30D";  // As if fixed<2,1>
```

Note that for initialization of strings, the digits and scale of the value are set precisely according to the rules in Section 4.21.4, whereas initialization from floating-point values may result in a much larger number of digits than you would expect, depending on how accurately a value can be represented as a floating-point number. For that reason, it is probably best to avoid initialization from floating-point numbers.

### 6.12.2 Accessors

The `fixed_digits` and `fixed_scale` member functions return the total number of digits and the number of fractional digits respectively:

```cpp
Fixed f = "3.14D";
cout << f.fixed_digits() << endl;  // Prints 3
cout << f.fixed_scale() << endl;   // Prints 2
```

### 6.12.3 Conversion Operators

The `LongLong` conversion operator converts a `Fixed` value back into a `LongLong` value, ignoring fractional digits. If the integral part of a `Fixed` value exceeds the range of `LongLong`, the operator throws a `DATA_CONVERSION` exception.

The `LongDouble` conversion operator converts a `Fixed` value to `LongDouble`.

### 6.12.4 Truncation and Rounding

The `truncate` member function returns a new `Fixed` value with the specified digits and scale, truncating fractional digits if necessary:

```cpp
Fixed f = "0.999";
cout << f.truncate(0) << endl;  // Prints 0
cout << f.truncate(1) << endl;   // Prints 0.9
cout << f.truncate(2) << endl;   // Prints 0.99
```
The `round` member function returns a new `Fixed` value with the specified digits and scale, rounded to the specified digit:

```cpp
Fixed r;

Fixed f1 = "0.4";
Fixed f2 = "0.45";
Fixed f3 = "-0.445";

r = f1.round(0); // 0
r = f1.round(1); // 0.5
r = f2.round(0); // 0
r = f2.round(1); // 0.5
r = f3.round(1); // -0.4
r = f3.round(2); // -0.45
```

Neither `truncate` nor `round` modifies the value it is applied to; instead, they return a new value.

### 6.12.5 Arithmetic Operators

The `Fixed` class provides the usual set of arithmetic operators. Arithmetic is carried out internally with at least 62-digit precision, and the result is coerced to fit a maximum of 31 digits, truncating fractional digits. If the result of an arithmetic operation exceeds 31 integral digits, arithmetic operators throw a `DATA_CONVERSION` exception.

### 6.12.6 Stream Operators

The `Fixed` mapping provides stream insertion (`<<`) and extraction (`>>`) operators. They work like their floating-point counterparts; that is, you can control padding and precision using the usual stream features.

### 6.13 Mapping for Structures

The C++ mapping treats fixed-length structures differently from variable-length structures, particularly with respect to parameter passing (see Section 7.14). We
first examine the mapping for fixed-length structures and then show the mapping and memory management rules for variable-length structures.

### 6.13.1 Mapping for Fixed-Length Structures

IDL structures map to C++ structures with corresponding members. For example:

```c++
struct Details {
    double weight;
    unsigned long count;
};
```

This IDL maps to

```c++
class Details_var;
```

```c++
struct Details {
    CORBA::Double weight;
    CORBA::ULong count;
    typedef Details_var _var_type;
    // Member functions here...
};
```

Note that the structure may have member functions, typically class-specific `operator new` and `operator delete`. These member functions allow use of the ORB on platforms that have non-uniform memory management. However, any additional member functions in the structure are purely internal to the mapping; you should ignore them and write your code as if they did not exist. The `_var_type` definition is used for template-based programming, and we show an example of its use in Section 18.14.1.

You can use the generated structure just as you use any other C++ structure in your code. For example:

```c++
Details d;
```

```c++
    d.weight = 8.5;
    d.count = 12;
```

C++ permits static initialization of aggregates. A class, structure, or array is an aggregate if it does not have user-declared constructors, base classes, virtual functions, or private or protected non-static data members. The preceding structure is an aggregate, so you can initialize it statically:

```c++
Details d = { 8.5, 12 };
Some C++ compilers have problems with aggregate initializations, so use the feature with caution.

### 6.13.2 Mapping for Variable-Length Structures

The `Details` structure shown in the preceding section is a fixed-length type, so there are no memory management issues to consider. For variable-length structures, the C++ mapping must deal with memory management. Here is an example:

```c++
struct Fraction {
    double numeric;
    string alphabetic;
};
```

This structure is a variable-length type because one of its members is a string. Here is the corresponding C++ mapping:

```c++
class Fraction_var;

struct Fraction {
    CORBA::Double numeric;
    CORBA::String_mgr alphabetic;
    typedef Fraction_var _var_type;
    // Member functions here...
};
```

As before, you can pretend that any member functions in the structure do not exist. As you can see, the IDL string is mapped to a type `String_mgr` instead of `String_var` or `char *`. `String_mgr` behaves like a `String_var` except that the default constructor initializes the string to the empty string instead of initializing it to a null pointer.

In general, strings nested inside user-defined types (such as structures, sequences, exceptions, and arrays) are always initialized to the empty string instead of to a null pointer. Initializing to the empty string for nested types is useful because it means that you need not explicitly initialize all string members inside a user-defined type before sending it across an IDL interface. (As you will see in Section 7.14.15, it is illegal to pass a null pointer across an IDL interface.)

---

2. Note that initialization to the empty string for nested string members was introduced with CORBA 2.3. In CORBA 2.2 and earlier versions, you must explicitly initialize nested string members.
If you look at the generated code for your ORB, you may find that the actual name of this class is something other than String_mgr, such as String_item or String_member. The exact name is not specified by the C++ mapping. For the remainder of this book, we use the name String_mgr whenever we show a string that is nested inside another data structure. A word of warning: do not use String_mgr (or its equivalent) as a type in your application code. If you do, you are writing non-portable code because the name of the type is not specified by the C++ mapping. Instead, always use String_var when you require a managed string type.

Apart from the initialization to the empty string, String_mgr behaves like a String_var. After you assign a string to the member alphabetic, the structure takes care of the memory management for the string; when the structure goes out of scope, the destructor for alphabetic deallocates its string for you. String_mgr provides the same conversions as String_var, and String_mgr and String_var can be freely assigned to each other, so you can effectively forget about the existence of String_mgr.

Automatic memory management is common to all structured types generated by the mapping. If a structure (or sequence, union, array, or exception) contains (perhaps recursively) a variable-length type, the structure takes care of the memory management of its contents. To you, this means that you need worry about the memory management only for the outermost type, and you need not worry about managing memory for the members of the type.

Here is an example to make this concept more concrete:

```c
{  
  Fraction f;
  f.numeric = 1.0/3.0;
  f.alphabetic = CORBA::string_dup("one third");
} // No memory leak here
```

Here, we declare a local variable f of type Fraction. The structure’s constructor performs memberwise initialization. For the member numeric, it does nothing. However, the member alphabetic is a nested string, so the constructor initializes it to the empty string.

The first assignment to the member numeric does nothing unusual. To assign to alphabetic, we must allocate memory, and alphabetic takes responsibility for deallocating that memory again (the assignment invokes operator=(char *) on alphabetic).

When f goes out of scope, its default destructor uses memberwise destruction and calls the destructor of alphabetic, which in turn calls
This means that there is no memory leak when \( f \) goes out of scope.

Note that you cannot statically initialize \( f \), because it is not a C++ aggregate (it contains a member with a constructor):

\[
\text{Fraction } f = \{ 1.0/3.0, \text{ "one third" } \}; \quad \text{// Compile-time error}
\]

In general, variable-length structures can never be statically initialized, because they contain members that have constructors.

### 6.13.3 Memory Management for Structures

You can treat structures in much the same way that you treat any other variable in your program. Most of the memory management activities are taken care of for you. This means that you can freely assign structures and structure members to one another:

```c
{
    struct Fraction f1;
    struct Fraction f2;
    struct Fraction f3;

    f1.numeric = .5;
    f1.alphabetic = CORBA::string_dup("one half");
    f2.numeric = .25;
    f2.alphabetic = CORBA::string_dup("one quarter");
    f3.numeric = .125;
    f3.alphabetic = CORBA::string_dup("one eighth");

    f2 = f1; \quad \text{// Deep assignment}
    f3.alphabetic = f1.alphabetic; \quad \text{// Deep assignment}
    f3.numeric = 1.0;
    f3.alphabetic[3] = \'\0\'; \quad \text{// Does not affect } f1 \text{ or } f2
    f1.alphabetic[0] = \'O\'; \quad \text{// Does not affect } f2 \text{ or } f3
    f1.alphabetic[4] = \'H\'; \quad \text{// Ditto}
} \quad \text{// Everything deallocated OK here}
```
Figure 6.2. Structures before and after assignments.

Figure 6.2 shows the initial and final values of the three structures for this example. As you can see, structure and member assignments make deep copies. Moreover, when the structures are deleted, the memory held by the three string members is automatically deallocated by the corresponding String_mgr destructor.

If you need to work with dynamically allocated structures, you use `new` and `delete`:

```cpp
Fraction * fp = new Fraction;
fp->numeric = 355.0 / 113;
fp->alphabetic = CORBA::string_dup("Pi, approximately");
// ...
delete fp;
```

There is no need to call special helper functions for allocation and deallocation. If such functions are required for non-uniform memory architectures, they are generated as class-specific `operator new` and `operator delete` members of the structure.

### 6.13.4 Structures Containing Structure Members

Structure members that are themselves structures do not require any special mapping rules:

```cpp
struct Fraction {
    double numeric;
    string alphabetic;
};

struct Problem {
```
string expression;
Fraction result;
boolean is_correct;
};

This generates the following mapping:

struct Fraction {
    CORBA::Double numeric;
    CORBA::String_mgr alphabetic;
    // ...
};

struct Problem {
    CORBA::String_mgr expression;
    Fraction result;
    CORBA::Boolean is_correct;
    // ...
};

Using a variable of type Problem follows the usual rules for initialization and assignment. For example:

Problem p;
p.expression = CORBA::string_dup("7/8");
p.result.numeric = 0.875;
p.result.alphabetic = CORBA::string_dup("seven eighths");
p.is_correct = 1;

Problem * p_ptr = new Problem;
*p_ptr = p; // Deep assignment
//
// It would be more efficient to use
// Problem * p_ptr = new Problem(p); // (deep) copy constructor
//
delete p_ptr; // Deep deletion

6.14 Mapping for Sequences

The mapping for sequences is large, mainly because sequences permit you to control allocation and ownership of the buffer that holds sequence elements. We discuss simple uses of unbounded sequences first and then show how you can use more advanced features to efficiently insert and extract data. The advanced features are particularly useful if you need to transmit binary data as an octet
sequence. Finally, we explain the mapping for bounded sequences, which is a subset of the mapping for unbounded sequences.

6.14.1 Mapping for Unbounded Sequences

IDL sequences are mapped to C++ classes that behave like vectors with a variable number of elements. Each IDL sequence type results in a separate C++ class. For example:

typedef sequence<string> StrSeq;

This maps to C++ as follows:

class StrSeq_var;

class StrSeq {
    public:
        StrSeq();
        StrSeq(CORBA::ULong max);
        StrSeq(
            CORBA::ULong max,
            CORBA::ULong len,
            char ** data,
            CORBA::Boolean release = 0
        );
        ~StrSeq();

        StrSeq(const StrSeq &);
        operator=(const StrSeq &);

        CORBA::String_mgr & operator[](CORBA::ULong idx);
        const char * operator[](CORBA::ULong idx) const;

        CORBA::ULong length() const;
        void length(CORBA::ULong newlen);
        CORBA::ULong maximum() const;

        CORBA::Boolean release() const;

        void replace(
            CORBA::ULong max,
            CORBA::ULong length,
            char ** data,
            CORBA::Boolean release = 0
        );
const char ** get_buffer() const;
char ** get_buffer(CORBA::Boolean orphan = 0);
static char ** allocbuf(CORBA::ULong nelems);
static void freebuf(char ** data);

typedef StrSeq_var _var_type;
};

This class is complicated. To get through all the definitions without too much pain, we discuss basic usage first and then cover the more esoteric member functions.3

StrSeq()
The default constructor creates an empty sequence. Calling the length accessor of a default-constructed sequence returns the value 0. The internal maximum of the sequence is set to 0 (see page 184).

StrSeq(const StrSeq &)
StrSeq & operator=(const StrSeq &)
The copy constructor and assignment operator make deep copies. The assignment operator first destroys the target sequence before making a copy of the source sequence (unless the release flag is set to false; see page 187). If the sequence elements are variable-length, the elements are deep-copied using their copy constructor. The internal maximum of the target sequence is set to the same value as the internal maximum of the source sequence (see page 184).

~StrSeq()
The destructor destroys a sequence. If the sequence contains variable-length elements, dynamic memory for the elements is also released (unless the release flag is set to false; see page 187).

CORBA::ULong length() const
The length accessor simply returns the current number of elements in the sequence.

3. The _var_type definition generated into the class is useful for template-based programming. We show an example in Section 18.14.1.
void length(CORBA::ULong newlen)

The length modifier changes the length of the sequence.

- Increasing the length of a sequence creates `newlen - length()` new elements. The new elements are appended to the tail. Growing a sequence initializes the newly appended elements with their default constructor. (If the appended elements are strings or are complex types containing strings, the strings are initialized to the empty string.)

- Decreasing the length of a sequence truncates the sequence by destroying the `length() - newlen` elements at the tail. If you truncate a sequence by reducing its length, the truncated elements are permanently destroyed. You cannot expect the previously truncated elements to still be intact after you increase the length again.

CORBA::String_mgr & operator[](CORBA::ULong idx)
const char * operator[](CORBA::ULong idx) const

The subscript operators provide access to the sequence elements (the operator is overloaded to allow use of sequence elements in both rvalue and lvalue contexts). In this example, using a sequence of strings, the return values are `String_mgr` and `const char *`, respectively. In general, for a sequence containing elements of type `T`, these operators return values of type `T &` and `const T &`, respectively. You may find that the actual type is something other than a reference to a `T`, depending on exactly how your ORB implements sequences. However, whatever type is returned, it will behave as if it were a reference to a `T`.

Sequences are indexed from 0 to `length() - 1`. Attempts to index into a sequence beyond its current length result in undefined behavior, and many ORBs will force a core dump to alert you of this run-time error.

If you do not like this, consider the alternatives: either you can run on blindly, happily corrupting memory as you go, or the ORB could throw an exception when a sequence index is out of bounds. However, that would not do you much good. After all, indexing a sequence out of bounds is a serious run-time error (just as overrunning an array is). What would be the point of throwing an exception? None—it would just tell you that you have a bug in your code.

**Simple Use of Sequences**

The few member functions we have just discussed are sufficient to make use of sequences. The following example demonstrates use of a sequence. The string elements behave like `String_mgr` instances:
const char * values[] = { "first", "second", "third", "fourth" };

StrSeq myseq; // Create empty sequence

// Create four empty strings
myseq.length(4);
for (CORBA::ULon i = 0; i < myseq.length(); i++)
    myseq[i] = values[i]; // Deep copy

// Print current contents
for (CORBA::ULon i = 0; i < myseq.length(); i++)
    cout << "myseq[" << i << "] = " << myseq[i] << "\n" << endl;

// Change second element (deallocates "second")
myseq[1] = CORBA::string_dup("second element");

// Truncate to three elements
myseq.length(3); // Deallocates "fourth"

// Grow to five elements (add two empty strings)
myseq.length(5);

// Initialize appended elements
myseq[3] = CORBA::string_dup("4th");
myseq[4] = CORBA::string_dup("5th");

// Print contents once more
for (CORBA::ULon i = 0; i < myseq.length(); i++)
    cout << "myseq[" << i << "] = " << myseq[i] << "\n" << endl;

This code produces the following output:

myseq[0] = "first"
myseq[1] = "second"
myseq[2] = "third"
myseq[3] = "fourth"

myseq[0] = "first"
myseq[1] = "second element"
myseq[2] = "third"
myseq[3] = "4th"
myseq[4] = "5th"

Once myseq goes out of scope, it invokes the destructor for its elements, so all the strings in the sequence are deallocated properly.
To manage heap-allocated sequences, use `new` and `delete`:

```cpp
StrSeq * ssp = new StrSeq;
ssp->length(4);
for (CORBA::ULong i = 0; i < ssp->length(); i++)
    (*ssp)[i] = values[i];
// ...
delete ssp;
```

If special allocation rules apply for non-uniform memory architectures, the sequence class contains appropriate class-specific allocation and deallocation operators.

You may be worried by the expression

```cpp
(*ssp)[i] = values[i];
```

Dereferencing the pointer is necessary, because we need an expression of type `StrSeq` for the subscript operator. If we instead write

```cpp
ssp[i] = values[i]; // Wrong!!!
```

the compiler assumes that we are dealing with an array of sequences and are assigning a `const char *` to the i-th sequence, which causes a compile-time error.

### Controlling the Sequence Maximum

When you construct a sequence variable, you can supply an anticipated maximum number of elements using the maximum constructor:

```cpp
StrSeq myseq(10); // Expect to put ten elements on the sequence
myseq.length(20); // Maximum does *not* limit length of sequence
for (CORBA::ULong i = 0; i < myseq.length(); i++)
    // Initialize elements
```

As you can see, even though this code uses an anticipated maximum of 10 elements, it then proceeds to add 20 elements to the sequence. This is perfectly all right. The sequence extends the maximum as necessary to accommodate the additional elements.

Why bother with supplying an anticipated maximum? The answer has to do with how a sequence manages its buffer space internally. If you use the maximum constructor, the sequence sets an internal maximum to a value at least as large as the one you supply (the actual maximum may be set to a larger value than the one you supply). In addition, a sequence guarantees that elements will not be relocated in memory while the current length does not exceed the maximum.
Typically, you do not care about relocation of elements in memory unless you are maintaining pointers to the sequence elements. In that case, you must know when sequence elements may relocate in memory because relocation will invalidate your pointers.

Another reason for supplying a maximum is efficiency. If the sequence has some idea of the expected number of elements, it can chunk memory allocations more efficiently. This approach reduces the number of calls to the memory allocator and reduces the number of times elements need to be copied as the sequence grows in length. (Memory allocation and data copying are expensive.)

You can retrieve the current maximum of a sequence by invoking the `maximum` member function. The following small program appends octets to a sequence one octet at a time and prints the maximum every time it changes:

```c++
int
main()
{
    BinaryFile s(20); // IDL: typedef sequence<octet> BinaryFile;

    CORBA::ULong max = s.maximum();
    cout << "Initial maximum: " << max << endl;

    for (CORBA::ULong i = 0; i < 256; i++) {
        s.length(i + 1);
        if (max != s.maximum()) {
            max = s.maximum();
            cout << "New maximum: " << max << endl;
        }
        s[i] = 0;
    }
    return 0;
}
```

On a particular ORB, this code might produce the following output:

```
Initial maximum: 64
New maximum: 128
New maximum: 192
New maximum: 256
```

This output allows you to reverse-engineer some knowledge about the sequence’s internal implementation. In this particular implementation, the sequence uses chunked allocation of 64 elements at a time, so the maximum of 20 given to the constructor is rounded up to 64. Thereafter, the sequence extends its internal
buffer space by another 64 elements whenever the length is incremented beyond a multiple of 64.

The same code, when run on a different ORB, might produce this output:

```
Initial maximum: 20
New maximum: 21
New maximum: 22
New maximum: 23
...
New maximum: 255
New maximum: 256
```

In this implementation, the sequence simply allocates buffer space as needed for each element.

For both implementations, whenever the maximum value changes, the actual octets may be relocated in memory, but they also may stay where they are, depending on the sequence implementation and the specific memory allocator in use.

Be careful not to interpret too much into the maximum constructor and the behavior of sequences.

- The mapping does not guarantee that the maximum constructor will preallocate memory at the time it is called. Instead, allocation may be delayed until the first element is created.
- The mapping does not guarantee that the maximum constructor will allocate memory for exactly the requested number of elements. It may allocate more.
- The mapping does not guarantee that the maximum constructor will use a single allocation to accommodate the requested number of elements. It may allocate sequence elements in several discontiguous buffers.
- The mapping does not guarantee that sequence elements occupy a contiguous region of memory. To avoid the cost of relocating elements, the sequence may add new discontiguous buffer space as it is extended.
- The mapping does not guarantee that extending the length of a sequence immediately default-constructs the newly created elements. Although this would be far-fetched, the mapping implementation could delay construction until a new element is first assigned to and at that point create the element using its copy constructor.

It should be clear that the maximum constructor is no more than a hint to the implementation of the sequence. If you create a sequence and have advance knowledge of the expected number of elements, then by all means, use the
maximum constructor. It may help to get better run-time performance from the sequence. Otherwise, do not bother.

Do not maintain pointers to sequence elements. If you do, you need to be extremely careful about reallocation. Usually, the trouble is not worth it.

**Using the Data Constructor**

The data constructor allows you to assign a preallocated buffer to a sequence. The main use of the data constructor is to efficiently transmit binary data as an octet sequence without having to use bytewise copying. There are a number of problems associated with the data constructor, and we recommend that you do not use it unless you have an overriding reason; you may wish to skip this section and continue reading on page 194. Still, we describe the data constructor for completeness.

The signature of the data constructor depends on the sequence element type. For example, for the sequence of strings shown on page 180, the signature is as follows:

```c++
StrSeq( // IDL: typedef sequence<string> StrSeq;
    CORBA::ULong max,
    CORBA::ULong len,
    char ** data,
    CORBA::Boolean release = 0
);
```

On the other hand, for a sequence of octets, the data constructor’s signature becomes

```c++
BinaryFile( // IDL: typedef sequence<octet> BinaryFile;
    CORBA::ULong max,
    CORBA::ULong len,
    CORBA::Octet * data,
    CORBA::Boolean release = 0
);
```

Note that the `data` parameter is of type pointer to element. The idea is that you can provide a pointer to a buffer full of elements and have the sequence use that buffer for its internal storage. To see why this may be useful, consider the following scenario.

Imagine you have a GIF image in a file and want to transmit that image to a remote server. The file contents are binary and need to get to the server without being tampered with in transit, so you decide to send the image as an octet sequence:
**Basic IDL-to-C++ Mapping**

```cpp
typedef sequence<octet> BinaryFile;

interface BinaryFileExchange {
    void send(in BinaryFile f, in string file_name);
    BinaryFile fetch(in string file_name);
};
```

On a UNIX system, a simple version of the code to initialize the sequence for transmission might look something like this (for simplicity, we have omitted error checking):

```cpp
int fd;
fd = open("image.gif", O_RDONLY);       // Open file for reading
struct stat st;
fstat(fd, &st);                         // Get file attributes
CORBA::Octet * buf;
buf = new CORBA::Octet[st.st_size];     // Allocate file buffer
read(fd, buf, st.st_size);             // Read file contents

BinaryFile image_seq(st.st_size);      // Create octet sequence
image_seq.length(st.st_size);         // Set length of sequence

// Fill sequence
for (off_t i = 0; i < st.st_size; i++)
    image_seq[i] = buf[i];

delete[] buf;                          // Don't need buffer anymore
close(fd);                             // Done with file

// Send octet sequence to server...
```

The image file might be several hundred kilobytes long, but the preceding code copies the file contents into the octet sequence one byte at a time. Even if the sequence’s subscript operator is inlined, this approach is still massively inefficient.

We can avoid this problem by using the data constructor:

```cpp
// Open file and get attributes as before...
CORBA::Octet * buf;
buf = new CORBA::Octet[st.st_size];     // Allocate file buffer
read(fd, buf, st.st_size);             // Read file contents
```

---

4. A word of caution here: sending a binary file as shown will not work once the file size exceeds an ORB-dependent limit. We discuss how to get around this in Section 18.7.
close(fd); // Done with file

// Initialize sequence with buffer just read
BinaryFile image_seq(st.st_size, st.st_size, buf, 0);

// Send octet sequence to server...
delete[] buf; // Deallocate buffer

The interesting line here is the call to the data constructor:

BinaryFile image_seq(st.st_size, st.st_size, buf, 0);

This call initializes both the maximum and the length of the sequence to the size
of the file, passes a pointer to the buffer, and sets the release flag to false. The
sequence now uses the passed buffer for its internal storage, thereby avoiding the
cost of initialising the sequence one byte at a time. Setting the release flag to
false indicates that we want to retain responsibility for memory management of
the buffer. The sequence does not deallocate the buffer contents. Instead, the
preceding code does this explicitly by calling delete[] when the sequence
contents are no longer needed.

If you set the release flag to true, the sequence takes ownership of the
passed buffer. In that case, the buffer must have been allocated with allocbuf,
and the sequence deallocates the buffer with freebuf:

// Open file and get attributes as before...
CORBA::Octet * buf;
buf = BinaryFile::allocbuf(st.st_size); // Allocate file buffer
read(fd, buf, st.st_size); // Read file contents

// Initialize, sequence takes ownership
BinaryFile image_seq(st.st_size, st.st_size, buf, 1);

close(fd); // Done with file

// Send octet sequence to server...

// No need to deallocate buf here, the sequence
// will deallocate it with BinaryFile::freebuf()

The allocbuf and freebuf member functions are provided to deal with non-
uniform memory architectures (for uniform architectures, they are simply imple-
mented in terms of new[] and delete[]). The allocbuf function returns a
null pointer if it fails to allocate memory (it does not throw C++ or CORBA
exceptions). It is legal to call freebuf with a null pointer.
If you initialize a sequence with `release` set to true as shown earlier, you cannot make assumptions about the lifetime of the passed buffer. For example, a compliant (although inefficient) implementation may decide to immediately copy the sequence and deallocate the buffer. This means that after you have handed the buffer to the sequence, the buffer becomes private memory that is completely out of your control.

If the `release` flag is true and the sequence elements are strings, the sequence will release memory for the strings when it deallocates the buffer. Similarly, if the release flag is true and the sequence elements are object references, the sequence will call `CORBA::release` on each reference.

String elements are deallocated by a call to `CORBA::string_free`, so you must allocate them with `CORBA::string_alloc`. The following example shows use of a sequence of strings with the `release` flag set to true. The code reads lines of text from a file, making each line a sequence element. Again, for brevity, we have not included any error handling. (The code also causes lines longer than 512 characters to be split, which we will assume is acceptable.)

```cpp
char linebuf[512]; // Line buffer
CORBA::ULong len = 0; // Current sequence length
CORBA::ULong max = 64; // Initial sequence max
char ** strvec = StrSeq::allocbuf(max); // Allocate initial chunk

ifstream infile("file.txt"); // Open input file

infile.getline(linebuf, sizeof(linebuf)); // Read first line
while (infile) { // While lines remain
    if (len == max) {
        // Double size if out of room
        char ** tmp = StrSeq::allocbuf(max *= 2);
        memcpy(tmp, strvec, len * sizeof(*strvec));
        StrSeq::freebuf(strvec);
        strvec = tmp;
    }
    strvec[len++] = CORBA::string_dup(linebuf); // Copy line
    infile.getline(linebuf, sizeof(linebuf)); // Read next line
}

StrSeq line_seq(max, len, strvec, 1); // Initialize seq
// From here, line_seq behaves like an ordinary string sequence:
for (CORBA::ULong i = 0; i < line_seq.length(); i++)
```
cout << line_seq[i] << endl;

line_seq.length(len + 1); // Add a line
line_seq[len++] = CORBA::string_dup("last line");

line_seq[0] = CORBA::string_dup("first line"); // No leak here

This example illustrates the memory management rules. The buffer that is eventually handed over to the string sequence is strvec. This buffer is initialized by a call to StrSeq::allocbuf, with sufficient room to hold 64 strings. During the loop reading the file, the code checks whether the current maximum has been reached; if it has, the code doubles the maximum (this requires reallocating and copying the vector). Each line is copied into the vector by CORBA::string_dup. When the loop terminates, strvec is a dynamically allocated vector of pointers in which each element points at a dynamically allocated string. This vector is finally used to initialize the sequence with the release flag set to true, so the sequence assumes ownership of the vector.

Once the sequence is initialized in this way, it behaves like an ordinary string sequence; that is, the elements are of type String_mgr, and they manage memory as usual. Similarly, the sequence can be extended or shortened and will take care of allocating and deallocating memory as appropriate.

Contrast this with a string sequence that is initialized with release set to false:

// Assume that:
// argv[0] == "a.out"
// argv[1] == "first"
// argv[2] == "second"
// argv[3] == "third"
// argv[4] == "fourth"
{
    StrSeq myseq(5, 5, argv); // release flag defaults to 0
    myseq[3] = "3rd"; // No deallocation, no copy
    cout << myseq[3] << endl; // Prints "3rd"
}

Because the release flag is false, the sequence uses shallow pointer assignment; it neither releases the target string "third" nor makes a copy of the source string "3rd". When the sequence goes out of scope, it does not release the
string vector, so the assignment’s effect is visible beyond the lifetime of the sequence.

Be careful, though: assignment to a sequence element is not guaranteed to affect the original vector. By slightly modifying the preceding code, we get different behavior:

```cpp
// Assume that:
// argv[0] == "a.out"
// argv[1] == "first"
// argv[2] == "second"
// argv[3] == "third"
// argv[4] == "fourth"
{
    StrSeq myseq(5, 5, argv); // release flag defaults to 0
    myseq[3] = "3rd"; // No deallocation, no copy
    cout << myseq[3] << endl; // Prints "3rd"
    myseq.length(10000); // Force reallocation
    myseq[1] = "1st"; // Shallow assignment
    cout << myseq[1] << endl; // Prints "1st"

    // deallocate whatever memory was allocated by length(10000)

    cout << argv[1] << endl; // prints "first" (not "1st")
    cout << argv[3] << endl; // prints "3rd"
}
```

This example uses two assignments to sequence elements but separates them by a large increase in the length of the sequence. This increase in length is likely to cause reallocation. (It is not guaranteed to force reallocation. An implementation is free instead to allocate additional separate memory while keeping the original vector, even though such an implementation is unlikely.) The effect is that the first assignment (before reallocation) affects the original vector, but the second assignment (after reallocation) affects only an internal copy, which is deallocated when the sequence goes out of scope.

This example demonstrates that initializing a sequence with `release` set to `false` requires a lot of caution. Unless you are very careful, you will leak memory or lose the effects of assignments.

Never pass a sequence with `release` set to `false` as an `inout` parameter to an operation. Although the called operation can find out how the sequence was allocated, it will typically assume that `release` is set to true. If the actual sequence has `release` set to `false`, assignment to sequence elements by the called operation can result in deallocation of non-heap memory, typically causing a core dump.
Manipulating the Sequence Buffer Directly

As you saw on page 180, sequences contain member functions to manipulate the buffer of a sequence directly. For the BinaryFile sequence, the generated code contains the following:

```cpp
class BinaryFile {
public:
    // Other member functions here...
    void replace(
        CORBA::ULong max,
        CORBA::ULong length,
        CORBA::Octet * data,
        CORBA::Boolean release = 0
    );

    const CORBA::Octet * get_buffer() const;
    CORBA::Octet * get_buffer(CORBA::Boolean orphan = 0);
    CORBA::Boolean release() const;
};
```

These member functions let you directly manipulate the buffer underlying a sequence.

The replace member function permits you to change the contents of a sequence by substituting a different buffer. The meaning of the parameters is the same as that for the data constructor. Obviously, the same caveats apply here as for shortening or lengthening of a sequence: if you are holding pointers into a sequence buffer and replace the buffer, the pointers are likely to point at garbage afterward.

The get_buffer accessor function provides read-only access to the underlying buffer. (If you call get_buffer on a sequence that does not yet have a buffer, the sequence allocates a buffer first.) The get_buffer function is useful for efficient extraction of sequence elements. For example, you can extract a binary file without copying the sequence elements:

```cpp
BinaryFile bf = ...;  // Get an image file...
CORBA::Octet * data = bf.get_buffer();  // Get pointer to buffer
CORBA::ULong len = bf.length();  // Get length
display_gif_image(data, len);  // Display image
```

This code obtains a pointer to the sequence data and passes the pointer to a display routine. The advantage here is that you can display the sequence contents without copying any elements.

The get_buffer modifier function provides read-write access to a sequence buffer. Its orphan argument determines who gets ownership of the
buffer. If orphan is false (the default), the sequence retains ownership and releases the buffer when it goes out of scope. If orphan is true, you become responsible for the returned buffer and must eventually deallocate it using freebuf.

You need to exercise caution if you decide to use the get_buffer modifier. The modifier enables you to assign to sequence elements in place. However, if the elements are strings, wide strings, or object references, you need to check the release flag of the sequence (returned by the release member function). If the release flag is false, you must not deallocate elements before assigning to them. If the release flag is true, you must deallocate sequence elements before assigning to them. The deallocation functions are CORBA::string_free, CORBA::wstring_free, and CORBA::release, depending on whether the sequence elements are strings, wide strings, or object references. (Other element types require no memory management from you.)

After you have taken ownership of the buffer from a sequence, the sequence reverts to the same state it would have if it had been constructed by its default constructor. If you attempt to remove ownership of a buffer from a sequence whose release flag is false, get_buffer returns a null pointer.

### 6.14.2 Mapping for Bounded Sequences

The mapping for bounded sequences is identical to the mapping for unbounded sequences except that the maximum is hard-wired into the generated class. For example:

```cpp
typedef sequence<double, 100> DoubleSeq;
```

This results in the following class:

```cpp
class DoubleSeq_var;

class DoubleSeq {
public:
    DoubleSeq();
    DoubleSeq(
        CORBA::ULong len,
        CORBA::Double * data,
        CORBA::Boolean release = 0
    );
    ~DoubleSeq();
    DoubleSeq(const DoubleSeq &);
};
```
6.14 Mapping for Sequences

As you can see, the only differences between a bounded sequence and an unbounded sequence are that for a bounded sequence, the maximum constructor is missing and that the data constructor does not accept a maximum parameter. (The maximum value of 100 is generated into the source code for the class.)

Attempts to set the length of a bounded sequence beyond the maximum result in undefined behavior, usually a core dump. Calls to allocbuf need not specify a number of elements that is the same as the sequence bound.

6.14.3 Sequence Limitations

Insertion and Deletion of Elements

An annoying aspect of the sequence mapping is that you can change the length of a sequence only at its tail. To insert an element somewhere in the middle, you must open a gap by copying the elements to the right of the insertion point. The following helper function preinserts an element into a sequence at a nominated position. Passing an index value equal to the length of the sequence appends the element at the tail. The function assumes that only legal index values in the range 0 to length() will be passed:
template<class Seq, class T>
void
pre_insert(Seq & seq, const T & elmt, CORBA::ULong idx)
{
    seq.length(seq.length() + 1);
    for (CORBA::ULong i = seq.length() - 1; i > idx; i--)
        seq[i] = seq[i - 1];
    seq[idx] = elmt;
}

This code extends the sequence by one element, opens a gap by copying elements from the insertion point to the tail over by one position, and then assigns the new element.

Similar code is required for removal of an element, in which you need to close the gap that is left behind at the deletion point:

template<class Seq>
void
remove(Seq & seq, CORBA::ULong idx)
{
    for (CORBA::ULong i = idx; i < seq.length() - 1; i++)
        seq[i] = seq[i + 1];
    seq.length(seq.length() - 1);
}

Insertion and removal operations on sequences have $O(n)$ run-time performance. This performance becomes unacceptable if frequent insertions or deletions are made, particularly for long sequences with elements of complex type. In such a case, you are better off using a more suitable data structure instead of trying to manipulate sequence elements in place.

For example, you can use an STL set or multiset to perform insertions and deletions in $O(\log n)$ time. After the set is in its final state, simply create an equivalent sequence by copying the contents of the set in a single pass. This technique is particularly useful if you need to make many updates to a sequence but want to keep the sequence in sorted order.

Using the Data Constructor with Complex Types

The data constructor is of limited value if a sequence contains elements of user-defined complex type. Consider the following IDL:

typedef string Word;
typedef sequence<Word> Line;
typedef sequence<Line> Document;
This IDL represents a line of text as a sequence of words, and a document as a sequence of lines. The problem for the data constructor is that we have no idea how the C++ class for a sequence of words is represented internally. For example, the sequence class will almost certainly have private data members that point at the dynamic memory for the sequence buffer. It follows that we cannot write a sequence value into a binary file and read the file later to reconstruct the sequence. By the time the file is read, the private pointer values of the sequence will likely point at the wrong memory locations.

You can use the sequence data constructor to create a sequence of complex values, but the sequence elements of the vector must be created by memberwise assignment or copy. For example:

```cpp
Line * docp = Document::allocbuf(3); // Three-line document
Line tmp; // Temporary line

tmp.length(4); // Initialize first line
tmp[0] = CORBA::string_dup("This");
tmp[1] = CORBA::string_dup("is");
tmp[2] = CORBA::string_dup("line");
tmp[3] = CORBA::string_dup("one.");
docp[0] = tmp; // Assign first line

tmp.length(1); // Initialize second line
tmp[0] = CORBA::string_dup("Line2");
docp[1] = tmp; // Assign second line

tmp[0] = CORBA::string_dup("Line3"); // Initialize third line
docp[2] = tmp; // Assign third line

Document my_doc(3, 3, docp, 1); // Use data constructor
// ...
```

This code is correct, but use of the data constructor no longer offers any advantage in performance (because the sequence elements cannot be created by reading them from a binary file or by copying memory). For this reason, you should avoid using the data constructor for anything except sequences of simple types and for sequences of string literals with the release flag set to false.

### 6.14.4 Rules for Using Sequences

Here are some rules for safe use of sequences.
• Do not make assumptions about when constructors or destructors run. The implementation of the sequence mapping is free to delay construction or destruction of elements for efficiency reasons. This means that your code must not rely on side effects from construction or destruction. Simply assume that elements are copy-constructed during the first assignment, default-constructed during the first access, and destroyed when a sequence is shortened or goes out of scope. In that way, you will not get any unpleasant surprises.

• Never pass a sequence to a function for modification if the release flag is false. If the sequence does not own its buffer, the called function will most likely cause memory leaks if it modifies sequence elements.

• Avoid using the data constructor for elements of complex type. For complex types, the data constructor does not offer any advantages but makes the source code more complex.

• Remember that increasing the length of a sequence beyond the current maximum may cause relocation of elements in memory.

• Do not index into a sequence beyond the current length.

• Do not increase the length of a bounded sequence beyond its bound.

• Do not use the data constructor or the buffer manipulation functions unless you really need to. Direct buffer manipulation is fraught with potential memory management errors, and you should first convince yourself that any savings in performance justify the additional coding and testing effort.

### 6.15 Mapping for Arrays

IDL arrays map to C++ arrays of the corresponding element type. String elements are mapped to `String_mgr` (or some other type proprietary to the mapping implementation). The point is that string elements are initialized to the empty string but otherwise behave like a `String_var` (that is, manage memory). For example:

```cpp
typedef float FloatArray[4];
typedef string StrArray[15][10];

struct S {
    string s_mem;
    long l_mem;
};
typedef S StructArray[20];
```
This maps to C++ as follows:

```c++
typedef CORBA::Float FloatArray[4];
typedef CORBA::Float FloatArray_slice;
FloatArray_slice * FloatArray_alloc();
FloatArray_slice * FloatArray_dup(const FloatArray_slice *);
void FloatArray_copy(
    FloatArray_slice * to,
    const FloatArray_slice * from);
void FloatArray_free(FloatArray_slice *);

typedef CORBA::String_mgr StrArray[15][10];
typedef CORBA::String_mgr StrArray_slice[10];
StrArray_slice * StrArray_alloc();
StrArray_slice * StrArray_dup(const StrArray_slice *);
void StrArray_copy(
    StrArray_slice * to,
    const StrArray_slice * from);
void StrArray_free(StrArray_slice *);

struct S {
    CORBA::String_mgr s_mem;
    CORBA::Long l_mem;
};
typedef S StructArray[20];
typedef S StructArray_slice;
StructArray_slice * StructArray_alloc();
StructArray_slice * StructArray_dup(const StructArray_slice *);
void StructArray_copy(
    StructArray_slice * to,
    const StructArray_slice * from);
void StructArray_free(StructArray_slice *);
```

As you can see, each IDL array definition generates a corresponding array definition in C++. This means that you can use IDL array types just as you use any other array type in your code. For example:
Basic IDL-to-C++ Mapping

```cpp
FloatArray my_f = { 1.0, 2.0, 3.0 };  
my_f[3] = my_f[2];

StrArray my_str;  
my_str[0][0] = CORBA::string_dup("Hello"); // Transfers ownership  
my_str[0][1] = my_str[0][0]; // Deep copy

StructArray my_s;  
my_s[0].s_mem = CORBA::string_dup("World"); // Transfers ownership  
my_s[0].l_mem = 5;
```

To dynamically allocate an array, you must use the generated allocation and deallocation functions (use of `new[]` and `delete[]` is not portable):

```cpp
// Allocate 2-D array of 150 empty strings  
StrArray_slice * sp1 = StrArray_alloc();

// Assign one element  
sp1[0][0] = CORBA::string_dup("Hello");

// Allocate copy of sp1  
StrArray_slice * sp2 = StrArray_dup(sp1);

StrArray x;  // 2-D array on the stack  
StrArray_copy(&x, sp1); // Copy contents of sp1 into x

StrArray_free(sp2);  // Deallocate  
StrArray_free(sp1);  // Deallocate
```

The allocation functions return a null pointer to indicate failure and do not throw CORBA or C++ exceptions.

The allocation functions use the array slice type that is generated. The slice type of an array is the element type of the first dimension (or, for a two-dimensional array, the row type). In C++, array expressions are converted to a pointer to the first element and the slice types make it easier to declare pointers of that type. For an array type `T`, a pointer to the first element can be declared as `T_slice *`. Because IDL arrays map to real C++ arrays, you can also use pointer arithmetic to iterate over the elements of an array.

The `StrArray_copy` function deep-copies the contents of an array. Neither the source nor the target array need be dynamically allocated. This function effectively implements assignment for arrays. (Because IDL arrays are mapped to C++ arrays and C++ does not support array assignment, the mapping cannot provide an overloaded operator for array assignment.)
6.16 Mapping for Unions

IDL unions cannot be mapped to C++ unions; variable-length union members (such as strings) are mapped to classes, but C++ does not permit unions to contain class members with non-trivial constructors. In addition, C++ unions are not discriminated. To get around this, IDL unions map to C++ classes. For example:

```cpp
union U switch (char) {
  case 'L':
    long long_mem;
  case 'c':
  case 'C':
    char char_mem;
  default:
    string string_mem;
};
```

The corresponding C++ class has an accessor and a modifier member function for each union member. In addition, there are member functions to control the discriminator and to deal with initialization and assignment:

```cpp
class U_var;

class U {
  public:
    U();
    U(const U &);
    ~U();
    U & operator=(const U &);

corba::Char _d() const;
    void _d(corba::Char);

corba::Long long_mem() const;
    void long_mem(corba::Long);
corba::Char char_mem() const;
    void char_mem(corba::Char);
    const char * string_mem() const;
    void string_mem(char *);
    void string_mem(const char *);
    void string_mem(const corba::String_var &);

typedef U_var _var_type;
};
```
As with other IDL generated types, there may be additional member functions in the class. If there are, these functions are internal to the mapping implementation and you should pretend they do not exist.5

### 6.16.1 Union Initialization and Assignment

As with other complex IDL types, a union has a constructor, a copy constructor, an assignment operator, and a destructor.

\[ U() \]

The default constructor of a union performs no application-visible initialization of the class. This means that you must explicitly initialize the union before reading any of its contents. You are not even allowed to read the discriminator value of a default-constructed union.

\[ U(const \ U \&) \]
\[ U \& \ \operator\= (const \ U \&) \]

The copy constructor and assignment operator make deep copies, so if a union contains a string, the string contents are copied appropriately.

\[ \sim U() \]

The destructor destroys a union. If the union contains a variable-length member, the memory for that member is deallocated correctly. Destroying an uninitialized default-constructed union is safe.

### 6.16.2 Union Member and Discriminator Access

To activate or assign to a union member, you invoke the corresponding modifier member function. Assigning to a union member also sets the discriminator value. You can read the discriminator by calling the \_d member function. For example:

\[
U \ \text{my}_u; \quad // \ 'my_u' \ is \ not \ initialized
my_u.long_mem(99); \quad // \ Activate \ long_mem
assert(my_u._d() == 'L'); \quad // \ Verify \ discriminator
assert(my_u.long_mem() == 99); \quad // \ Verify \ value
\]

---

5. We delay explanation of the \_var_type definition in this class until Section 18.14.1, where we show an example of its use.
In this example, the union is not initialized after default construction. Calling the modifier function for the member `long_mem` initializes the union by activating that member and setting its value. As a side effect, assigning to a member via the modifier function also sets the discriminator value. The preceding code tests the discriminator value in an assertion to verify that the union works correctly. It also reads the value of `long_mem` by calling its accessor member function. Because we just set the value to 99, the accessor must of course return that value. The code tests this with another assertion.

To change the active member of a union, you can use the modifier for a different member to assign to that member:

```cpp
my_u.char_mem('X'); // Activate and assign to char_mem
// Discriminator is now 'c' or 'C', who knows...
my_u._d('C'); // Now it is definitely 'C'
```

Activating the member `char_mem` sets the discriminator value accordingly. The problem in this case is that there are two legal discriminator values: 'c' and 'C'. Activating the member `char_mem` sets the discriminator to one of these two values, but you have no way of knowing which one (the choice is implementation-dependent). The preceding code example explicitly sets the value of the discriminator to 'C' after activating the member.

You cannot set the discriminator value if that would deactivate or activate a member:

```cpp
my_u.char_mem('X'); // Activate and assign char_mem
assert(my_u._d() == 'c' || my_u._d() == 'C');
my_u._d('c'); // OK
my_u._d('C'); // OK
my_u._d('X'); // Illegal, would activate string_mem
```

The preceding example shows that you can set the discriminator only to a value that is consistent with the currently active union member (the only legal values here are 'c' and 'C'). Setting the discriminator value to anything else results in undefined behavior, and many implementations will deliberately force a core dump to let you know that your program contains a serious run-time error.

Setting the default member of the union leaves the discriminator in a partially undefined state:

```cpp
my_u.string_mem(CORBA::string_dup("Hello"));
// Discriminator value is now anything except 'c', 'C', or 'L'.
assert(my_u._d() != 'c' && my_u._d() != 'C' && my_u._d() != 'L');
```
The implementation of the union type picks a discriminator value that is legal for the default member, but, again, the precise value chosen is implementation-dependent.

This behavior can be inconvenient, for example during tracing. Suppose you have trace statements throughout your code that print the discriminator value to the display at various points. A problem arises if the default member string_mem is active in the union, because the value of the discriminator can be any character except 'c', 'C', and 'L'. This makes it entirely possible for the discriminator to contain non-printable characters, such as a form feed, escape, or Ctrl-S. Depending on the display you are using, these characters may cause undesirable effects. For example, an escape character can cause the display to clear its screen or switch into block mode, and a Ctrl-S typically acts as a flow-control character that suspends output.

In general, the default case and multiple case labels for the same union member do not assign a definite value to the discriminator of the union. We recommend that you use these IDL features with caution. Usually, you can express the desired design in some other way and avoid the potentially awkward coding issues involved.

The preceding example also illustrates another important point. String members inside a union behave like a String_var. In particular, the modifier function for the member string_mem is overloaded for const char *, char *, and String_var &. As always, the char * modifier takes ownership of the assigned string, whereas the const char * and String_var modifiers make deep copies:

```cpp
U my_u;

// Explicit copy
my_u.string_mem(CORBA::string_dup("Hello"));

// Free "Hello", copy "World"
my_u.string_mem((const char *)"World");

CORBA::String_var s = CORBA::string_dup("Again");
// Free "World", copy "Again"
my_u.string_mem(s);

// Free "Again", activate long_mem
my_u.long_mem(999);

cout << s << endl;    // Prints "Again"
```
For dynamically allocated unions, use `new` and `delete`:

```cpp
U * up = new U;
up->string_mem(CORBA::string_dup("Hello"));
// ...
delete up;
```

On architectures with non-uniform memory management, the ORB generates class-specific allocation and deallocation operators for the union, so you can still safely use `new` and `delete`.

### 6.16.3 Unions without a default Case

Here is a union that can be used to simulate optional parameters (see page 67):

```cpp
union AgeOpt switch (boolean) {
  case TRUE:
    unsigned short age;
};
```

This union does not have an explicit default case but has an implicit default member when the discriminator is FALSE. If a union has an implicit default member, the mapping generates an additional `_default` member function for the corresponding C++ class:

```cpp
class AgeOpt_var;

class AgeOpt {
  public:
    AgeOpt();
    AgeOpt(const AgeOpt &);
    ~AgeOpt();
    AgeOpt & operator=(const AgeOpt &);
    CORBA::Boolean _d() const;
    void _d(CORBA::Boolean);
    CORBA::UShort age() const;
    void age(CORBA::UShort);
    void _default();

typedef AgeOpt_var _var_type;
};
```
The mapping follows the normal rules but also adds the _default member function. (It is a little unfortunate that a union without a default case has an extra member function called _default. You have to get used to this.) The _default member function activates the implicit default member of the union and sets the discriminator value accordingly:

```cpp
AgeOpt my_age;
my_age._default(); // Set discriminator to false
```

In this case, the only legal default value for the discriminator is 0 (which represents false). Note that the following code is illegal:

```cpp
AgeOpt my_age;
my_age._d(0); // Illegal!
```

This code has undefined behavior, because it is illegal to activate a union member by setting the discriminator. (The non-existent implicit default member of the union is considered a member.)

Similarly, you cannot reset an initialized union to the default member by setting the discriminator. You must instead use the _default member function:

```cpp
AgeOpt my_age;
my_age.age(38); // Sets discriminator to 1
my_age._d(0); // Illegal!!!
my_age._default(); // Much better!
```

Here is another interesting union, taken from the Trading Service Specification [21]:

```cpp
eenum HowManyProps { none, some, all };

union SpecifiedProps switch (HowManyProps) {
    case some:
        PropertyNameSeq prop_names;
};
```

This union permits two different discriminator values for the no-value case: none and all. Suppose you want to initialize the union to set the discriminator value to none. Again, you must use the _default member function:

```cpp
SpecifiedProps sp;
sp._default(); // Activate implicit default member
sp._d(none); // Discriminator is now none or all
```

The call to _default is necessary. Without it, we would attempt to activate the implicit default member by setting the discriminator, and that is illegal.
6.16.4 Unions Containing Complex Members

If a union contains a member that is of type any or contains a member that is a structure, union, sequence, or fixed-point type, the generated class contains three member functions for each union member instead of the usual two member functions. Consider the following union:

```cpp
struct Details {
    double weight;
    long count;
};

typedef sequence<string> TextSeq;

union ShippingInfo switch (long) {
    case 0:
        Details packaging_info;
    default:
        TextSeq other_info;
};
```

This union has two members: one is a structure and the other one is a sequence. The generated class contains all the member functions we discussed previously but has three member functions for each union member:

```cpp
class ShippingInfo {
    public:
        // Other member functions as before...

        const Details & dl_mem() const; // Accessor
        void dl_mem(const Details &); // Modifier
        Details & dl_mem(); // Referent

        const TextSeq & seq_mem() const; // Accessor
        void seq_mem(const TextSeq &); // Modifier
        TextSeq & seq_mem(); // Referent
};
```

As with simple types, the union contains accessor functions that return the value of a member. (To avoid unnecessary data copying, accessors for complex types return the value by constant reference.) Also, as with simple types, each member has a modifier function that makes a deep copy.

The referent member function returns a non-constant reference to the union member and exists for efficiency reasons. For large types, such as sequences, it is inefficient to change a member by calling its accessor followed by its modifier,
because both functions make deep copies. The referent permits you to modify the value of a union member in place without copying:

```cpp
ShippingInfo info = ...; // Assume we have an initialized union...

if (info._d() != 0) { // other_info is active
    TextSeq & s = info.other_info(); // get ref to other_info
    // We can now modify the sequence while it is
    // inside the union without having to copy
    // the sequence out of the union and back in again...
    for (CORBA::ULong i = 0; i < s.length(); i++) {
        // Modify sequence elements...
    }
}
```

Of course, if you obtain a reference to a union member, that member must currently be active (otherwise the behavior is undefined). Once you have a reference to a member, you must take care to use it only for as long as its corresponding member remains active. If you activate a different union member and use a reference to a previously active member, you are likely to end up with a core dump.

### 6.16.5 Rules for Using Unions

Here are some rules for using unions safely.

- Never attempt to access a union member that is inconsistent with the discriminator value. This is just common sense. Unions are not meant to be used as a backdoor mechanism for type casts. To safely read the value of a union member, first check the discriminator value. It is common to check the discriminator in a switch statement and to process each union member in a different branch of the switch. Be careful if you obtain a reference to a union member. The reference stays valid only for as long as its member remains active.

- Do not assume that union members overlay one another in memory. In C and C++, you are guaranteed that union members overlay one another in memory. However, no such guarantee is provided by the C++ mapping for IDL unions. A compliant ORB may keep all union members active simultaneously, or it may overlay some union members but not others. This behavior allows the ORB to intelligently adjust the behavior of a union depending on its member
6.17 Mapping for Recursive Structures and Unions

Consider the following recursive union:

```c++
union Link switch (long) {
    case 0:
        typeA ta;
    case 1:
        typeB tb;
    case 2:
        sequence<Link> sc;
};
```

The union contains a recursive member `sc`. Assume that you would like to activate the `sc` member of this union so that `sc` is an empty sequence. As you saw earlier, the only way to activate a union member is to pass a value of the member’s type to its accessor. However, `sc` is of anonymous type, so how can you declare a variable of that type?

The C++ mapping deals with this problem by generating an additional type definition into the union class:

```c++
class Link {
public:
    typedef some_internal_identifier _sc_seq;

    // Other members here...
};
```

The generated class defines the type name `_sc_seq` to give a name to the otherwise anonymous type. In general, if a union `u` contains a member `mem` of
anonymous type, the type of `mem` has the name `u::_mem_seq`. You can use this type name to correctly activate the recursive member of a union:

```cpp
Link::sc_seq myseq;       // myseq is empty
Link mylink;              // uninitialized union
mylink.sc(myseq);         // activate sc
```

The same mapping rule applies to recursive structures. If a structure `s` contains an anonymous sequence member `mem`, the type of `mem` is `s::_mem_seq`.

### 6.18 Mapping for Type Definitions

IDL type definitions map to corresponding type definitions at the C++ level. If a single IDL type results in multiple C++ types, each C++ type has a corresponding type definition. Aliasing of type definitions is preserved. If function declarations are affected by aliasing, a corresponding function using the alias name is defined (usually as an inline function):

```cpp
typedef string StrArray[4];
typedef StrArray Address;
```

This definition maps as follows:

```cpp
typedef CORBA::String_mgr StrArray[4];
typedef CORBA::String_mgr StrArray_slice;
StrArray_slice * StrArray_alloc();
StrArray_slice * StrArray_dup(const StrArray_slice *);
void StrArray_free(StrArray_slice *);

typedef StrArray Address;
typedef StrArray_slice Address_slice;

Address_slice * Address_alloc()
    { return StrArray_alloc(); }

Address_slice * Address_dup(
    const Address_slice * p
    ) { return StrArray_dup(p); }

void Address_free(Address_slice * p)
    { StrArray_free(p); }
```

The preceding code looks complicated, but it really means that aliases for types can be used in exactly the same way as the original type. For example, with the
preceding mapping, you can use StrArray and Address interchangeably in your code.

### 6.19 User-Defined Types and _var Classes

As shown earlier in Table 6.2 on page 155, the IDL compiler generates a _var class for every user-defined structured type. These _var classes serve the same purpose as String_var; that is, they take on memory management responsibility for a dynamically allocated instance of the underlying type.

Figure 6.3 shows the general idea of the generated _var class for an IDL type T, where T is a structure, union, or sequence. An instance of a _var class holds a private pointer to an instance of the underlying type. That instance is assumed to be dynamically allocated and is deallocated by the destructor when the _var instance goes out of scope.

The _var class acts as a smart pointer that wraps the underlying type. The overloaded indirection operator delegates member function calls on the _var instance to the underlying instance. Consider the following code fragment, which assumes that T is a sequence type:

```cpp
class T_var {
public:
    T_var();
    T_var(T *);
    T_var(const T_var &);
    ~T();
    T_var & operator=(T *);
    T_var & operator=(const T_var &);
    T * operator->();
    const T * operator->() const;
    // etc...
private:
    T * myT;
};
```

```cpp
class T { // or struct T
public:
    // Public members of T...
};
```

**Figure 6.3.** _var class for structures, unions, and sequences.
This example illustrates that instances of a _var class behave much like ordinary C++ class instance pointers. The difference is that _var classes also manage memory for the underlying type.

### 6.19.1 _var Classes for Structures, Unions, and Sequences

The following code shows the general form of _var classes for structures, unions, and sequences. (Depending on the exact underlying type, there may be additional member functions, which we discuss shortly.)

```cpp
class T_var {
public:
    T_var();
    T_var(T *);
    T_var(const T_var &);
    ~T_var();

    T_var & operator=(T *);
    T_var & operator=(const T_var &);

    T * operator->();
    const T * operator->() const;

    T & operator[](CORBA::ULong); // For sequences
    const T & operator[](CORBA::ULong) const; // For sequences

    // Other member functions here...
private:
    T * myT;
};
```

The default constructor initializes the internal pointer to the underlying instance to null. As a result, you cannot use a default-constructed _var instance until after you have initialized it.
T_var(T *)
The pointer constructor assumes that the passed pointer points to a dynamically
allocated instance and takes ownership of the pointer.

T_var(const T_var &)
The copy constructor makes a deep copy of both the T_var and its underlying
instance of type T. This means that assignment to a copy-constructed T_var
affects only that copy and not the instance it was copied from.

~T_var()
The destructor deallocates the instance pointed to by the internal pointer.

T_var & operator=(T *)
The pointer assignment operator first deallocates the instance of type T currently
held by the target T_var and then assumes ownership of the instance pointed to
by its argument.

T_var & operator=(const T_var &)
The T_var assignment operator first deallocates the instance of type T currently
held by the target T_var and then makes a deep assignment of both the T_var
argument and the instance of type T that the argument points to.

T * operator->() const
The indirection operator is overloaded to permit its use on both constant and non-
constant instances of the underlying type. It returns a pointer to the underlying
instance. This means that you can use the T_var to invoke any member function
of the underlying type.

operator T &() const
These conversion operators permit a T_var to be used in places where a constant
or non-constant reference to the underlying type is expected.
T & operator[](CORBA::ULong)
const T & operator[](CORBA::ULong) const

The subscript operators are generated if the T_var represents a sequence or an array. They permit you to index into a sequence as if the T_var were the actual sequence or array type. The operators exist for convenience, letting you avoid awkward expressions such as sv->operator[](0).

### 6.19.2 Simple Use of _var Classes

Let us consider a simple example of using the _var class for a sequence. The IDL definition of the sequence is

```idl
typedef sequence<string> NameSeq;
```

This generates two C++ types: NameSeq, which is the actual sequence, and NameSeq_var, which is the corresponding memory management wrapper. Here is a code fragment that illustrates use of NameSeq_var instances:

```c++
NameSeq_var ns; // Default constructor
ns = new NameSeq; // ns assumes ownership
ns->length(1); // Create one empty string
ns[0] = CORBA::string_dup("Bjarne"); // Explicit copy

NameSeq_var ns2(ns); // Deep copy constructor
ns2[0] = CORBA::string_dup("Stan"); // Deallocates "Bjarne"

NameSeq_var ns3; // Default constructor
ns3 = ns2; // Deep assignment
ns3[0] = CORBA::string_dup("Andrew"); // Deallocates "Stan"

cout << ns[0] << endl; // Prints "Bjarne"
cout << ns2[0] << endl; // Prints "Stan"
cout << ns3[0] << endl; // Prints "Andrew"

// When ns, ns2, and ns3 go out of scope,
// everything is deallocated cleanly...
```

As with String_var, the generated _var types are useful mainly to catch return values for dynamically allocated variable-length types. For example:

```c++
extern NameSeq * get_names(); // Returns heap-allocated instance
NameSeq_var nsv = get_names(); // nsv takes ownership
// No need to worry about deallocation from here on...
```
As you will see in Section 7.14, such allocation frequently happens when a client invokes an IDL operation. Using a _var instance to take ownership means that you need not constantly remember to deallocate the value at the correct time.

### 6.19.3 Some Pitfalls of Using _var Classes

Similar caveats apply to generic _var classes as apply to String_var. If you initialize a _var instance with a pointer or assign a pointer, you need to make sure that the pointer really points at dynamically allocated memory. Failure to do so results in disaster:

```c++
NameSeq names; // Local sequence
// ...
// Initialize sequence
NameSeq_var nsv(&names); // Looming disaster!
NameSeq_var nsv(new NameSeq(names)); // Much better!
```

After you have assigned a pointer to a _var instance, you must be careful when dereferencing that pointer:

```c++
NameSeq_var famous = new NameSeq;
famous->length(1);
famous[0] = CORBA::string_dup("Bjarne");
NameSeq * fp = famous; // Shallow assignment
NameSeq * ifp;
{
    NameSeq_var infamous = new NameSeq;
    infamous->length(1);
    infamous[0] = CORBA::string_dup("Bill");
    ifp = infamous; // Shallow assignment
    famous = infamous; // Deep assignment
}
cout << (*fp)[0] << endl; // Whoops, fp points nowhere
cout << (*ifp)[0] << endl; // Whoops, ifp points nowhere
```

These problems arise because assignment to a _var deallocates the previous underlying instance and so invalidates a pointer still pointing to that instance. Similarly, when a _var instance goes out of scope, it deallocates the underlying instance and invalidates any pointers still pointing at that instance.

In practice, such problems rarely occur because _var classes are used mainly to avoid memory leaks for return values and out parameters. You will see more examples of using _var classes in Section 7.14.12.
6.19.4 Differences Among Fixed- and Variable-Length Structures, Unions, and Sequences

The generated _var classes vary slightly in their interfaces depending on whether they wrap a fixed-length or a variable-length type. Normally, these differences are transparent to you. They exist to hide differences in parameter passing rules for fixed-length and variable-length types (we discuss this in more detail in Section 7.14.12).

All _var classes provide in, inout, out, and _retn member functions (with different signatures depending on whether the _var class wraps a variable- or a fixed-length type). In addition, _var classes for variable-length types have an extra conversion operator, whereas _var classes for fixed-length types provide an extra constructor and assignment operator.

Additional T_var Member Functions for Variable-Length Types

In addition to the member functions discussed on page 212, for a variable-length structure, union, or sequence of type T, the IDL compiler generates the following:

class T_var {
    public:
        // Normal member functions here...

        // Member functions for variable-length T:
        operator T * &();
        const T & in() const;
        T & inout();
        T * & out();
        T * _retn();
};

operator T * &()

This additional conversion operator allows you to pass a variable-length T_var where a reference to a pointer to T is expected. This operator is used if T_var instances for variable-length types are passed as out parameters. We discuss this in detail in Section 7.14.

class T_var {
    public:
        // Normal member functions here...

        // Member functions for variable-length T:
        operator T * &();
        const T & in() const;
        T & inout();
        T * & out();
        T * _retn();
};

operator T * &()

This additional conversion operator allows you to pass a variable-length T_var where a reference to a pointer to T is expected. This operator is used if T_var instances for variable-length types are passed as out parameters. We discuss this in detail in Section 7.14.
These member functions allow you to explicitly pass a T_var as an in, inout, or out parameter instead of relying on default conversions. The functions are useful mainly if your compiler has defects relating to default conversions. You can also call these functions explicitly to improve code readability. If you pass a T_var instance to a function, it may not be immediately obvious whether the called function will modify the underlying value. By using these member functions, you can improve readability of the code:

```cpp
StrSeq_var sv = ...;  
some_func(sv);    // Passed as in, inout, or out?
some_func(sv.out());   // Much clearer...
```

The `out` member function deallocates the underlying instance of type T as a side effect to prevent memory leaks if the same T_var instance is passed to successive calls:

```cpp
StrSeq_var sv = ...;  
some_func(sv.out());   // Sets sv to heap-allocated instance.
some_func(sv.out());   // Deallocates previous instance, assumes
                        // ownership of new instance.
```

T * _retn

This function returns the pointer to the underlying instance of type T and also relinquishes ownership of that pointer. It is useful mainly when you create a T_var to avoid memory leaks but then must transfer ownership of the underlying type (see page 168 for an example).

**Additional T_var Member Functions for Fixed-Length Types**

For a T_var for a fixed-length structure, union, or sequence of type T, the IDL compiler generates the following:

```cpp
class T_var {
public:
   // Normal member functions here...

   // Member functions for fixed-length T:
   T_var(const T &);
   T_var & operator=(const T &);
   const T & in() const;
   T & inout();
   T & out();
   T _retn();
};
```
T_var(const T &)
T & operator=(const T &)

The additional constructor and assignment operator permit you to construct or
assign a T_var from a T.

const T & in() const
T & inout()
T & out()
T _retn()

These member functions are provided to deal with defective compilers that cannot
handle default conversions correctly. They also make the direction in which a
parameter is passed explicit at the point of call, something that improves code
readability.

The out and _retn member functions for fixed-length types do not relinquish ownership of the underlying type. They cannot do this because they do not
return a pointer.

6.19.5 _var Types for Arrays

The _var types generated for arrays follow a similar pattern as those for struc
tures, unions, and sequences. The differences are that _var types for arrays do
not overload the indirection operator (it is not needed for arrays) and that the
return types of some of the member functions are different. _var types for arrays
with variable-length and fixed-length elements also have some differences.

Array _var Mapping for Arrays with Variable-Length Elements

It is easiest to illustrate the mapping with an example. Here we define a three-
element array containing variable-length structures:

```cpp
struct Fraction { // Variable-length structure
    double numeric;
    string alphabetic;
};
typedef Fraction FractArr[3];
```

This maps to the following C++ definitions:

```cpp
struct Fraction {
    CORBA::Double numeric;
    CORBA::String_mgr alphabetic;
};
```
class Fraction_var {
public:
    // As before...
};

typedef Fraction FractArr[3];
typedef Fraction FractArr_slice;

FractArr_slice * FractArr_alloc();
FractArr_slice * FractArr_dup(const FractArr_slice *);
void FractArr_copy(
    FractArr_slice * to,
    const FractArr_slice * from
);
void FractArr_free(FractArr_slice *);

class FractArr_var {
public:
    FractArr_var();
    FractArr_var(FractArr_slice *);
    FractArr_var(const FractArr_var &);
    ~FractArr_var();

    FractArr_var & operator=(FractArr_slice *);
    FractArr_var & operator=(const FractArr_var & rhs);
    Fraction & operator[](CORBA::ULong);
    const Fraction & operator[](CORBA::ULong) const;

    operator FractArr_slice *();
    operator const FractArr_slice *() const;
    operator FractArr_slice * &();

    const FractArr_slice * in() const;
    FractArr_slice * inout();
    FractArr_slice * & out();
    FractArr_slice * _retn();
};

If all this looks a little intimidating, remember that the various member functions do exactly the same things as for _var types for structures, unions, and sequences.

* The default constructor initializes the internal pointer to the underlying array to null.
• Constructors and assignment operators that accept an argument of type FractArr_slice * assume that the array was allocated with FractArr_alloc or FractArr_dup, and they take ownership of the passed pointer.
• The copy constructor and FractArr_var & assignment operator each make a deep copy.
• The destructor deallocates the array by calling FractArr_free.
• The subscript operators allow indexing into the array, so you can use a FractArr_var as if it were the actual array.
• The conversion operators permit passing the array as an in, inout, or out parameter (see Section 7.14.12).
• The explicit conversion functions in, inout, and out behave as for structures, unions, and sequences (see page 216).
• The _retn function permits you to relinquish ownership of the underlying type (see page 168 for an example).

All this means that you can use an array _var as if it were the actual array; you just need to remember that an array _var must be initialized with dynamically allocated memory.

const char * fractions[] = { "1/2", "1/3", "1/4" };

FractArr_var fa1 = FractArr_alloc();
for (CORBA::ULon i = 0; i < 3; i++) { // Initialize fa1
    fa1[i].numeric = 1.0 / (i + 2);
    fa1[i].alphabetic = fractions[i]; // Deep copy
}

FractArr_var fa2 = fa1; // Deep copy
fa2[0].alphabetic = CORBA::string_dup("half"); // Explicit copy
fa2[1] = fa2[2]; // Deep assignment

cout.precision(2);
for (CORBA::ULong i = 0; i < 3; i++) { // Print fa1
    cout << "fa1[" << i << "]\.numeric = "
    << fa1[i].numeric
    << ",\tfal[" << i << "]\.alphabetic = "
    << fa1[i].alphabetic << endl;
}

fa1 = fa2; // Deep assignment
fa1[0].numeric = 0.0;
for (CORBA::ULong i = 0; i < 3; i++) { // Print fa1
    cout << "fa1[" << i << "]\.numeric = "
    << fa1[i].numeric
    << ",\tfal[" << i << "]\.alphabetic = "
    << fa1[i].alphabetic << endl;
}

fa1 = fa2; // Deep assignment
fa1[0].numeric = 0.0;
for (CORBA::ULong i = 0; i < 3; i++) { // Print fa2
    cout << "fa2[" << i << "].numeric = "
    << fa2[i].numeric
    << ",\tfal[" << i << "].alphabetic = "
    << fa2[i].alphabetic << endl;
}
<< fa2[i].numeric
<< ",\tfa2[" << i << "]\t.alphabetic = "
<< fa2[i].alphabetic << endl;
}

The output of this program is as follows:

fa1[0].numeric = 0.5, fa1[0].alphabetic = 1/2
fa1[1].numeric = 0.33, fa1[1].alphabetic = 1/3
fa1[2].numeric = 0.25, fa1[2].alphabetic = 1/4

fa2[0].numeric = 0.5, fa2[0].alphabetic = half
fa2[1].numeric = 0.25, fa2[1].alphabetic = 1/4
fa2[2].numeric = 0.25, fa2[2].alphabetic = 1/4

**Array _var Mapping for Arrays with Fixed-Length Elements**

The mapping for _var types for arrays with fixed-length elements is almost identical to the mapping for _var types for arrays with variable-length elements. Here we define a three-element array containing fixed-length structures:

```cpp
struct S { // Fixed-length structure
    long l_mem;
    char c_mem;
};
typedef S StructArray[3];
```

The mapping for the corresponding StructArray_var type is as follows:

```cpp
class StructArray_var {
public:
    StructArray_var();
    StructArray_var(StructArray_slice *);
    StructArray_var(const StructArray_var &);
    ~StructArray_var();

    StructArray_var & operator=(StructArray_slice *);
    StructArray_var & operator=(const StructArray_var & rhs);

    S & operator[](CORBA::ULong);
    const S & operator[](CORBA::ULong) const;

    operator StructArray_slice *();
    operator const StructArray_slice *() const;

    const StructArray_slice * in() const;
};
```
StructArray_slice * inout();
StructArray_slice * out();
StructArray_slice * _retn();

The only differences between _var types for arrays with fixed-length and those for variable-length elements are that for fixed-length elements, the inout member function returns a pointer instead of a reference to a pointer and that no user-defined conversion operator for StructArray_slice * & is defined. These differences originate in the different parameter passing rules for variable-length and fixed-length types. We discuss these rules in detail in Section 7.14.

6.20 Summary

The basic C++ mapping defines how built-in types and user-defined types map to C++. Although some of the classes generated by the mapping have a large number of member functions, within a short time you will find yourself using them as you use any other data type. Even the memory management rules, which may seem complex right now, soon become second nature. When writing your code, keep in mind that you should be looking at the IDL definitions and not at the generated header files. In that way, you avoid getting confused by many internal details and cryptic work-arounds for different platforms and compilers.