Language System Release Notes

SPARCompiler™ C++

Version 3.0

SunPro

A Sun Microsystems, Inc. Business

SPARCompiler C++ 3.0.1 Language System

Release Notes

A Sun Microsystems, Inc. Business

2550 Garcia Avenue
Mountain View, CA 94043
U.S.A.

Part No: 800-6988-11
Revision A, October 1992

Contents

Preface

The *C++ 3.0.1 Language System Release Notes* describes Release 3.0 of the C++ Language System. The manual is part of a set of four documents that are supplied with your C++ Language System. The other documents are:

- The *C++ 3.0.1 Language System Product Reference Manual,* which provides a complete definition of the C++ language supported by Release 3.0.1 of the C++ Language System
- The *C++ 3.0.1 Language System Selected Readings,* which contains papers that describe aspects of the C++ language
- The *C++ 3.0.1 Language System Library Manual,* which describes the three C++ class libraries and tells you how to use them.

This *Release Notes* consists of two chapters and three appendices, that describe how to use the translator, changes in the C++ language for this release, and other information you need to know.

- Chapter 1, "Introduction," is a general description of the C++ Language System and new features that are part of this release. You should read this chapter as a general introduction to the release.
- Chapter 2, "Compatibility," describes compatibility between different releases of the C++ Language System; it describes things that have changed and might require you to make changes in code written for previous releases. This chapter discusses the following release changes:
  - Upgrading from Release 2.0 or Release 2.1 to Release 3.0.1.

- Future compatibility — changes and enhancements that are planned for the next major release of the C++ Language System This chapter contains detailed discussions of new features and changes included in the release, and, as such, should be an important reference for all users.

- Appendix A, "Known Problems," describes known problems with the C++ Language System which are of general interest to C++ programmers, and suggests workarounds for these problems.
- Appendix B, "Implementation Specific Behavior," describes implementation specific behavior.
- Appendix C, "Not Implemented Messages," is a list of "not implemented" messages issued by Release 3.0.1.

To make the best use of the *Release Notes*, you must be familiar with the C programming language and the C programming environment under the UNIX operating system.

Introduction

The C++, Release 3.0.1, translates C++ source code to object code by first translating the source into intermediate C code and then invoking the C compiler to create an object file. The CC command invokes the `acpp`, `cfront`, `acomp`, `fbe` and `ld` programs to translate the C++ program into the executable format. The CC command may also invoke the `ptcomp` and `ptlink` programs to handle templates.

New Features Introduced in Release 3.0.1

Release 3.0.1 is a release of C++, that is source– and link–compatible with Release 2.0 and Release 2.1. Release 3.0.1 provides the following new or enhanced features:

- The major feature for the release is Template classes and functions. For a definition and description of how to use this feature, please refer to Stroustrup, "Parameterized Types for C++" in the *C++ 3.0.1 Language System Selected Readings* as well as Chapter 14 of the *Reference Manual*, by Margaret Ellis and Bjarne Stroustrup (Addison-Wesley, 1990). This implementation conforms to the draft submitted to and preliminarily accepted by the ANSI C++ standards committee. However, users should note that there continues to be much discussion in the ANSI committee about the precise details of syntax and semantics regarding templates. While we do not expect major, incompatible changes in the definition of templates, it is likely that various refinements and extensions to the feature will be made in the course of the standardization activity. Users should be aware that any such refinements and extensions will be reflected in future releases of the AT&T USL C++

Language System. The Templates implementation is based on work originally done at Object Design Inc., in which they implemented template classes based on Stroustrup's initial design. We have licensed this initial templates implementation from Object Design and evolved it to include function templates, and have extended the class implementation to provide support for various language features such as friends and static members.

- Release 3.0.1 completes the implementation of true nested scopes introduced in Release 2.1. The transition model is no longer supported. Code that compiled warning-free under Release 2.1 will correctly reflect the new nested semantics.
- Release 3.0.1 begins a phased approach to improving the architecture of cfront. This release includes reworking of the front end symbol table, type checking, function matching, operator overloading and user-defined conversions.
- Release 3.0.1 implements various Release 2.1 *Reference Manual* upgrades, including allowing constructors in which all parameters have default arguments to be used as the default constructor in initializing arrays, overloaded prefix and postfix increment and decrement, extension of dominance to data, and use of constructor syntax for built-in types and protected derivations.
- Release 3.0.1 treats as errors most anachronisms which were warned about by default in Release 2.1. Those that were `+w` only warnings are generally being warned about by default in Release 3.0.1 and will be disabled in the release following 3.0.1.

Compatibility

2.1 Overview

This chapter describes compatibility issues that pertain to both Release 2.1 and Release 3.0.1. If you are currently using Release 2.1 and want to know about upgrading to Release 3.0.1, you can read the section "Upgrading from Release 2.1 to Release 3.0.1" below. If you are currently using Release 2.0, you should begin with "Upgrading from Release 2.0 to Release 2.1" on page 32. In either case you should also read the last section, "Future Compatibility Issues" on page 81, to learn about changes that will occur in the next major release of the C++ Language System.

2.2 Upgrading from Release 2.1 to Release 3.0.1

This section describes differences between Release 2.1 and Release 3.0.1. This section provides information on the following topics:

- New Features
- Language Related Fixes

Note – For further information on migration from release 2.1 to Release 3.0.1, see the *C++ 3.0.1 Programmer's Guide*, Appendix C, section C.3 "C++ 2.1 K&R and C++ 3.0.1 ANSI Differences" and the *SunOS Transition Guide*.

Recompilation of Release 2.1 Code

Code which compiled warning-free under Release 2.1 will not need to be recompiled. Code which uses nested types and which was not upgraded to use the transition model of Release 2.1 will need to be recompiled:

```
struct A {
      struct B {
            void f();
      };
};

typedef A::B T;

void T::foo() {};          // encoded as f__1BFv in 2.1
                           // encoded as f__Q2_1A1BFv in 3.0
```

However, code which used the new nesting semantics in Release 2.1 will continue to link correctly:

```
struct B {};        // force new nesting semantics
struct A {
      struct B {
            void f();
      };
};

typedef A::B T;

void T::f() {};     // encoded as f__Q2_1A1BFv in 2.1 and 3.0
```

Refer to the section below on Nested Types for further information.

New Features

The major new feature for the release is the implementation of Templates. Various new features introduced in the Release 2.1 Reference Manual have also been implemented.

Templates

The major enhancement in this release is the implementation of Templates. Both template classes and functions are supported. Automatic instantiation of templates is also provided. For a description of the feature and its uses, see Chapter 14 of the *C++ 3.0.1 Language System Product Reference Manual*, and "Parameterized Types for C++," B. Stroustrup, in the *C++ 3.0.1 Language System Selected Readings* manual. For information about support for automatic instantiation, refer to the *C++ 3.0.1 Language System Selected Readings*, Chapter 7, "Template Instantiation in C++ Release 3.0.1, Overview," G. McCluskey and R. B. Murray, which presents an overview and technical rationale for the instantiation mechanism and Chapter 8, "Template Instantiation, Users Guide," G. McCluskey, which presents various examples of use of the automated support for instantiations.

This implementation conforms to the draft submitted to and preliminarily accepted by the ANSI C++ standards committee. However, users should note that there continues to be much discussion in the ANSI committee about the precise details of syntax and semantics regarding templates. While we do not expect major, incompatible changes in the definition of templates, it is likely that various refinements and extensions to the feature will be made in the course of the standardization activity. Users should be aware that any such refinements and extensions will be reflected in future releases of the AT&T USL C++ Language System.

During AT&T USL C++ Release 3.0.1 beta testing, the restrictive function matching rules specified in the *C++ 3.0.1 Language System Product Reference Manual* were found to be too restrictive for practical use. We have, therefore, implemented extensions to the strict function matching rules in the *C++ 3.0.1 Language System Product Reference Manual*. It is likely that the ANSI definition will at least be relaxed to allow these extensions and may extend the definition to encompass full function matching. This is currently an active topic of discussion within the ANSI committee. In the meantime, we have made the smallest set of extensions we found feasible.

The first extension allows the consideration of trivial conversions when searching for an exact match. This implies that for a template function declared as follows:

```
template <class T> max( const T*, int );
```

the following call is legal in the Release 3.0.1 implementation:

```
int ia[10] = { ... };

// error in the Reference Manual
// accepted by 3.0
int best = max( ia, 10 );
```

The second extension allows the conversion of derived classes to public base classes in calls of template functions. This is necessary to ensure that template functions support object-oriented programming. For example, under the strict rules the following calls of function `print_vector()` fail:

```
template <class T> void print_vector(const Vector<T>&);

template <class T>
class BoundedVector : public Vector<T> { ... };

template <class T>
class SortedVector : public Vector<T> { ... };

BoundedVector<int> bv;
SortedVector<int> sv;

print_vector(bv);      // error in Reference Manual
                       // accepted by 3.0

print_vector(sv);      // error in Reference Manual
                       // accepted by 3.0
```

and separate print functions for each class derived from `Vector<T>` must be written. Permitting the conversions of `BoundedVector<int>` and `SortedVector<int>` to `Vector<int>` allows the use of the polymorphic `print_vector()` function.

These extensions are designed to be interim solutions until the ANSI committee votes on a full resolution to the template function matching issue.

An important aspect of the Release 3.0.1 implementation is support for automatic instantiation of template class and template function references. The Release 3.0.1 implementation provides an instantiation mechanism designed to

free the programmer from direct manual intervention. Manual overrides for complicated systems are provided to customize and tailor instantiation support for specialized applications. As discussed above, papers describing template instantiation are included in the *C++ 3.0.1 Language System Selected Readings* with this release. These papers make clear that some assumptions are made about coding style and conventions:

- A class or function template is declared in a `.h` header. For a function template this declaration looks like a forward function declaration:

  ```
  template <class T> void f(T);
  ```

  The template `.h` header should include headers, with multiple-include guards, for "unbound", i.e., non-template-arg types that it uses.

- A class or function template is implemented in a `.c` header.

- Template arguments of non-fundamental type are declared in `.h` header files. These files should be self-contained, i.e., include other files they need using multiple-include guards. Here is a simple example to get started with:

```
// Sample.h
template <class T> class Sample {
      char* p;
public:
      Sample(char* s) : p(s) {}
      char* get();
      char* get2() {return T::f();}
};

// Sample.c
template <class T> char* Sample<T>::get()
{
      return p;
}

// A.h
struct A {
      static char* f() {return " ";}
};

// application
#include <stdio.h>
#include "Sample.h"
#include "A.h"

Sample<A> a("Hello");

main()
{
     Sample<A> b("world");

     printf("%s%s%s", a.get(), a.get2(), b.get());
}
```

This is a complicated way of printing "Hello world." To compile this example, you would create the various files noted above and say:

```
$ CC app.c
```

It is instructive to look in the directory `./ptrepository` after such a compile. There are three files there whose use is fully explained in the paper:

- *xxx*`.c`- the instantiation file
- *xxx*`.o`- the instantiation itself
- *xxx*`.cs`- the checksum used for dependency management

Nested Types

Release 3.0.1 also completes the implementation of true nested scopes introduced in Release 2.1. The transition model is no longer supported. Code that compiled warning free under Release 2.1 will correctly reflect the new nested semantics. Please note that code that generated warnings under Release 2.1 may produce results under complete nested semantics that differ from Release 2.0 behavior:

```
class A {
       class B {
              ...
       };
       ...
};

B bvar;

      // 2.1: warning: use A:: to access nested class type B
      // (anachronism)

      // 3.0: error: B bvar : B is not a type name
      // error: type expected for bvar
```

Support for deeply nested classes is also now provided:

```
class A {
      class B {
            class C {
                  ...
            };
            ...
      };
};
```

Reference to the inner class C is now possible:

```
A::B::C cvar;
```

Default Constructors

The Release 2.0 *Reference Manual* explicitly stated that a default constructor is a constructor with no formal parameters, thereby excluding constructors that can be called with no arguments by virtue of having default arguments. The Release 2.1 and Release 3.0.1 versions of the *Reference Manual* lift this restriction; the constructor in the example below is now considered a default constructor.

```
struct S {
S(int = 0);
};
```

Release 2.1 does not conform to this rule. Instead, it adheres to the old definition of default constructor. Here are some examples:

```
S s1[2];              // legal, OK in 3.0, error in 2.1
S s2[2] = { 1 };      // legal, OK in 3.0, sorry in 2.1

struct X {
      S s[2];         // legal, OK in 3.0, error in 2.1
};

void f() {
      S* p = new S[2];  // legal, OK in 3.0 and 2.1
}
```

Release 3.0.1 correctly conforms to this rule.

Explicit Type Conversions with Empty Initializers

The Release 2.1 and 3.0.1 versions of the *Reference Manual* allow you to specify an explicit type conversion with an empty initializer, as in the following examples:

```
int i = int();

struct Empty {};
Empty e = Empty();
```

Release 2.1 does not implement this capability and reports an error instead.

```
line 1: error: value missing in conversion to int
line 4: error: cannot make a Empty
```

Release 3.0.1 implements this capability.

Prefix and Postfix Increment and Decrement Operators

The Release 2.0 *Reference Manual* provided no way to distinguish user-defined prefix increment and decrement operators from postfix increment and decrement operators. The Release 2.1 and 3.0.1 versions of the *Reference Manual*

specify a separate syntax for defining prefix and postfix increment and decrement operators. The prefix increment and decrement operators take one argument (the implicit `this` argument for a member function), whereas the postfix version takes two arguments (including the implicit `this` argument). For example,

```
struct S {
      operator++();           // 2.0: prefix or postfix
                            // 2.1: prefix, but not implemented as such
                             // 3.0: prefix, implemented as such
      operator++(int);      // 2.1: postfix ++, not implemented
                             // 3.0: postfix ++, implemented
};
```

However, Release 2.1 does not recognize the new syntax. Use of the postfix form results in the following error message:

```
line 4: error: S:: operator ++() takes no argument
```

Release 3.0.1 correctly handles these operators.

Extension of Dominance Rule to Objects

The Release 2.1 *Reference Manual* extended dominance to data and enumerators as well as functions. Release 2.1 did not implement this. Release 3.0.1 does:

```
enum E {a,b};
struct V {void f(); int x; E y;};
struct B: public virtual V {void f(); int x; E y;};
struct C: public virtual V{};

struct D: public B, public C {void g();};

void D::g() {
x++;                  // ambiguous in 2.0/2.1
                      // ok in 3.0, refers to B::x
y = a;                // ambiguous in 2.0/2.1
                      // ok in 3.0, refers to B::y
f();                  // ok in 2.0/2.1 and 3.0, refers to B::f
}
```

Protected Derivation

The Release 2.0 *Reference Manual* explicitly disallowed the use of `protected` as an access specifier for a base class. The Release 2.1 *Reference Manual* lifts this restriction. However, Release 2.1 does not implement the new behavior.

```
struct B {};
struct D : protected B {};// legal, but rejected by 2.1
// accepted by 3.0
```

Release 3.0.1 correctly implements the new behavior.

Exception Handling Syntax

Release 3.0.1 does not include an implementation of exception handling. However, the ANSI C++ committee has preliminarily accepted the exception handling scheme as described in Chapter 15 of *The Annotated C++ Reference Manual*. In Release 2.1, reserved words were added for exception handling. In Release 3.0.1, the likely syntax for exception handling has been incorporated into the grammar and a "sorry not implemented" message is generated for users. Again, code that compiled warning free under Release 2.1 will continue to compile and execute correctly under Release 3.0.1. Please note that Release 2.1 code that compiled with warnings about use of reserved words may result in surprising error messages under Release 3.0.1:

```
int try;
     // 2.1: warning: try is a future reserved keyword

     // 3.0: sorry, not implemented: try
     // error: syntax error
```

Language-Related Fixes

The focus of development for Release 3.0.1 has been to implement the Templates feature and to reengineer selected portions of the implementation. The reengineering focus has been on function matching, operator overloading, user-defined conversions, type checking and reworking the front end symbol table. We know of no bugs in the function matching or operator overloading and many of the scoping and name reuse bugs that existed in previous releases

have been fixed. The reworking of type checking has uncovered previously existing bugs that are now fixed. Please note, some of these fixes may change the behavior of programs for which Release 2.0 or Release 2.1 incorrectly accepted illegal code or produced incorrect results.

Section numbers (§) following a heading identify the section of the Release 3.0.1 *Reference Manual* that describes the correct behavior.

Declarations in `for` Initializers (§6.6, 6.8)

The Release 2.0 *Reference Manual* stated that a `for` statement containing a declaration in its *for-init-statement* was not allowed to be the statement after an `if`, `else`, `switch`, `while`, `do`, or `for`. In other words, this code was illegal:

```
void f(int i) {
      if (i)
            for (int j = i; j; j--)      // error
                  ;
}
```

This restriction was an error not enforced by the Release 2.0 implementation, and the Release 2.1 *Reference Manual* omits it.

The Release 2.1 *Reference Manual*, however, does specify a related restriction: "An `auto` variable constructed under a condition is destroyed under that condition and cannot be accessed outside that condition."

Here is an example:

```
int g(int i) {
      if (i)
           for (int j = 5; j; j--)
                 ;
      return j;// error
}
```

In the above code, `j` cannot be accessed at the point of the `return` statement because the `return` statement is outside the body of the `if` statement. According to the Release 2.1 *Reference Manual*, an error should be reported, but Release 2.1 quietly accepts this code. Release 3.0.1 correctly reports the error.

Enforcement of Return from Value-Returning Functions (§6.7)

In C++, unlike C, it is an error to fail to return a value from a value-returning function. See Section 6.6.3 of the *C++ 3.0.1 Language System Product Reference Manual*. Earlier releases of the compiler warned about failure to return a value. For Release 3.0.1, these warnings are errors for all member functions and all function templates. For non-member functions, failure to return a value when a return type is explicitly specified is an error; warnings will continue to be generated for non-member functions that implicitly return ints. As with previous releases, we will continue to warn about failure to return from `main` only under `+w`:

```
main() {/*...*/};                  // no return from main
                                   // +w warning in 2.0, 2.1 and 3.0

f() { /*... */ };                  // no return, implicit return type
                                   // warning in 2.0, 2.1 and 3.0

int f2() { /*...*/ };              // no return, explicit return type
                                   // warning in 2.0, 2.1
                                   // error in 3.0
struct A {
      f() {/*...*/};               // no return, implicit return type
                                   // warning in 2.0, 2.1
                                   // error in 3.0

      int f2() {/*...*/};          // no return, explicit return type
                                   // warning in 2.0, 2.1
                                   // error in 3.0
};
```

`const` Typedefs (§7.2)

Previous releases failed to unwind `const` typedefs correctly:

```
typedef char *T;

const char *p;        // p is a pointer to a const char
const T cp;           // cp is a constant pointer to char
```

Previous releases incorrectly evaluated `cp` as a pointer to `const char`.

Scope of a Class Member's Initializer (§8.5)

The Release 2.1 *Reference Manual* states explicitly that an initializer for a static member is in the scope of the member's class. This rule was not explicitly given in the previous *Reference Manual.*

Release 2.1 does not apply this rule consistently. For example, in

```
const int a = 5;

struct X {
      static int a;
      static int b;
};

int X::a = 1;
int X::b = a;
```

the correct behavior is implemented: `X::b` is initialized with `X::a`.

However, default arguments for member functions are not resolved within the scope of the class. In the following code,

```
const int y = 2;

struct Y {
      static int y;
      static int f(int);
};

int Y::f(int i = y) { return i; }
```

Release 2.1 incorrectly determines that the default argument for `Y::f()` is global `y`, not `Y::y`. Release 3.0.1 correctly resolves the argument.

Reference Initializers (§8.5)

The Release 2.0 *Reference Manual* allowed a reference to be initialized with a temporary, as in the following declaration:

```
int& r = 5;
```

However, the Release 2.1 *Reference Manual* has tightened the rules for reference initializations so that only `const` references may legally be initialized with non-lvalues. This means that, instead of the previous declaration, you must use the following:

```
const int& cr = 5;
```

The Release 2.0 C++ Language System already treated temporary initializers for non-`const` reference initializations at global scope as errors, although it allowed them at local scope. To provide a smooth transition to the more restrictive rules, Release 2.1 issues an anachronism warning, under control of the `+w` option, for non-`const` reference initializations that were accepted by Release 2.0 but are now illegal.

Here are some examples:

```
int& r1 = 5;          // illegal, error in 2.0, 2.1 and 3.0

struct A { A(int); ~A(); };
A& a1 = 5;           // illegal, sorry in 2.0, error in 2.1, 3.0
const A& a2 = 5;      // legal

int& f1();
int& r2 = f1();       // ok, 'f()' returns an lvalue

const int& r3 = 5;    // ok, 'r3' is 'const int&'

int f2(int&);
int j = f2(5);        // illegal, error in 2.0 and 2.1

void x() {
      int& r1 = 0;       // illegal, 2.1 warns under +w
                         // illegal, 3.0 warns by default
      A& a1 = 5;         // illegal, 2.1 warns under +w
                         // illegal, 3.0 warns by default
      const A& a2 = 5;   // legal, accepted by 2.0 and 2.1
      int j = f2(5);     // illegal, 2.0 and 2.1 warn under +w
                         // illegal, 3.0 warns by default
}


struct S1 {};
struct S2 {
      operator S1();
};

void f3(S1&);
void y(S2 s2) {
     f3(s2);             // illegal, 2.0 and 2.1 warn under +w
                  //illegal, 3.0 warns by default
}
```

Release 3.0.1 issues an unconditional warning, or an error if the +p option is in effect. The anachronism warnings turn into errors if the +p option is specified to the CC command.

Calls to Non-const Member Functions from const Objects (§9.4)

Calling a non-const member function on a const object has been illegal since Release 2.0. However, to ease transition to this new rule, calling a non-const member function on a const object was flagged with a warning in Release 2.0 and Release 2.1. This type of call is an error in Release 3.0.1. The obvious example of the effect of this change is the simple changing of a warning to an error as in the following case:

```
struct A {
    A();
    void foo();
};
const A a;
a.foo();      // a warning in 2.0 and 2.1
              // an error in 3.0
```

However, this change may also cause more subtle changes of behavior in code using function matching, operator overloading, or conversion functions. For example, a non-const member function is now eliminated from consideration in a call to an overloaded member function using a const object. For example:

```
struct A {
    A();
    void foo(int);              // #1
    void foo(char) const;       // #2
    void foo(const A*);         // #3
};
const A a;
a.foo(1);     // used to call #1 with warning
              // now will call #2
a.foo(&a);    // used to call #3 with warning
              // now flagged as no match error
```

Similarly, non-const user-defined operators are not considered for calls with const objects, and no non-const conversion operators will be applied to const objects.

An example with conversion operators:

```
class B {
    public:
        B();
        operator int();
}

const B b;
int i = b;      // error in 3.0
```

New errors that occur as a result of all usable functions being non-const should issue messages that include that information. For example, the program above gives the following error in Release 3.0.1:

```
"prog.c", line 8: error: bad initializer type const B for i (int
expected)
(no usable const conversion)
```

Enforcement of `const` in `const` Member Functions (§9.4)

As with calls to non-`const` member functions from `const` objects, the enforcement of `const` within `const` member functions was introduced via warnings in Release 2.0 and Release 2.1. In Release 3.0.1, the `const` rules are strictly enforced. The release correctly reports errors for assignment to data members or calls to non-`const` member functions from within a `const` member function. It is also illegal for `const` member functions to return non-

`const` references to a data member if the member is a class object. If the data member being returned is a built-in type, however, Release 3.0.1 still incorrectly reports this with just a warning.

```
struct B{ };

struct A {
      int i;
      B b;

      int& f() {return i;};          // ok, non-const member

      void f1(int j) const {
      i = j;                         // warning in 2.0/2.1


      i = f();                       // warning in 2.0/2.1
                                     // error in 3.0
      }

      int& f2() const {return i;}    // warning in 2.0/2.1

      B& f3() const {return b;}      // warning in 2.0/2.1/3.0
};                                   // error in 3.0
```

Static Data Members of Local Classes (§9.5)

The Release 2.1 *Reference Manual* states that static data members are not allowed for local classes. Previously, a local class could have a static data member only if no explicit initialization was required.

Release 2.1 does not enforce the new restriction properly. If a static data member of a local class is declared but never used, a warning is reported but the program links successfully

```
int main() {
      struct S {
            static int i;
      };
      // ...
      return 0;
}
```

```
line 2: warning: static member S::i in local class S (anachronism)
```

Release 3.0.1 enforces this restriction, and correctly reports an error.

Access Specifiers in Unions (§11.1)

The Release 2.1 *Reference Manual* allows access specifiers in unions. Formerly, these were forbidden.

```
union U {
public:                    // legal
      U();
      int i;
private:                   // legal
      double d;
protected:// legal
      float f;
};

U u;
float f = u.f;            // protection violation
```

Release 2.1 accepts the definition of U shown above but does not report the protection violation. Release 3.0.1 correctly flags the protection violation.

Access to Static Members of Private Base Classes (§11.3)

The Release 2.1 *Reference Manual* states that a private derivation of a base class does not restrict access to the static members of the base class. Without this rule, a member function would have less access to a base class's static members than a global function.

Release 2.1 does not implement this rule consistently. For access to a static member of an immediate base class, some illegal accesses are not reported:

```
struct B {
      static void f();
};

struct D : private B {}
struct E : private D {
      void g() {
            f();              // illegal, reported by 2.1 and 3.0
            this->f();        // illegal, reported by 2.1 and 3.0
            B::f();           // legal, OK in 3.0, rejected by 2.1
      }
};
```

In the above code, the calls `f()` and `this->f()` are illegal because they refer to `f()` via the `this` pointer, and thus the access protection for `private` members is applied. The call `B::f()` is legal because it refers to `f()` directly, just as a global function could refer to `B::f()`.

Release 3.0.1 enforces the rule consistently. If multi-level derivation is involved, both Releases 2.0 and 2.1 are overly conservative; they report an error for `X::f()` even though it is legal.

```
struct X {
        static void f();
};
struct Y : private X {};
struct Z : public Y {
        void g() {
                f();      // illegal, error in 2.0, 2.1 and 3.0
                this->f();// illegal, error in 2.0, 2.1 and 3.0
                X::f();   // legal, error in 2.0 and 2.1
        }
};
```

Scope of Friend Functions (§11.5)

The Release 2.1 *Reference Manual* states that a `friend` function defined within a class declaration is in the lexical scope of that class, just like a member function.

In general, Release 2.1 does not implement this rule. Consider the following example:

```
extern int s;
extern int e;

struct S {
      static int s;
      enum { e = 5 };
      friend f() { return e; }      // which 'e'?
      friend void g(int = s) { };   // which 's'?
};
```

According to the Release 2.1 *Reference Manual*, `f()` returns `S::e` and the default argument for `g()` is `S::s`. Instead, both Release 2.0 and 2.1 incorrectly resolve these names to `::e` and `::s` respectively. Release 3.0.1 resolves these names correctly.

Constructor and Destructor Declarations (§12.2, 12.5, 9.4)

The Release 2.1 *Reference Manual* specifies that constructors and destructors cannot be declared `const`, `volatile`, or `static`. Release 2.1 correctly reports an error for constructors and destructors that are declared `static`, but it

incorrectly allows constructors and destructors to be declared `const`. Release 2.1 does not implement `volatile` member functions at all; these are rejected with a "not implemented" message.

```
struct S {
      static S();           // illegal, error in 2.0, 2.1 and 3.0
      static ~S();          // illegal, error in 2.0, 2.1 and 3.0
};

struct T {
       T() const;            // illegal, but accepted by 2.1
                             // rejected by 3.0
~T() const;                  // illegal, but accepted by 2.1
                             // rejected by 3.0
T(char*) volatile;           // illegal, sorry in 2.1
                             // rejected by 3.0
};
```

Release 3.0.1 correctly reports these errors.

Destructors for Built-In Types (§12.5)

The Release 2.1 *Reference Manual* allows explicit destructor calls for any built-in type, as in the example below. However, Release 2.1 does not implement this syntax.

```
void f(int* p) {
    p->int::~int(); // legal, but error in 2.1
                    // legal, handled properly in 3.0
};
```

Release 3.0.1 correctly implements this syntax.

Delete Operator (§12.6)

The Release 2.1 *Reference Manual* tightens the rules for the `delete` operator. Only one `operator delete()` may be declared per class, and the global `operator delete()` may not be overloaded. Release 2.1 does not enforce these restrictions.

For example, the second declaration of the `delete` operator in each scope below is illegal, but the code is accepted by both Release 2.0 and 2.1.

```
typedef unsigned int size_t;

void operator delete(void*);
void operator delete(const void*);  // error, correctly reported
                                    // in 3.0

struct S {
      void* operator new(size_t);
      void* operator new(size_t, void*);
      void operator delete(void*);
    void operator delete(void*, size_t); // error, correctly
                                         //reported in 3.0
};
```

Release 3.0.1 correctly reports these errors.

Argument Matching Rules (§13.3)

Several details about the function matching rules have changed.

- In the Release 2.0 *Reference Manual* there was a rule that a call needing only standard conversions is preferred over one requiring user-defined conversions. This rule has been eliminated in the Release 2.1 *Reference Manual* and the new semantics have been implemented in Release 2.1. For example,

```
struct Complex { Complex(double); };
void f2(int, Complex);
void f2(double, double);

void y2() {
      f2(3, 4);// ambiguous
}
```

For this code, Release 2.1 and 3.0.1 correctly report an ambiguity.

- The second function matching change involves the treatment of arguments of type `T` that require temporaries. The Release 2.0 *Reference Manual* specified that a match with conversions requiring temporaries was a legal match. So, for example, the call to `f3(char&)` in the following code was legal and was accepted by Release 2.0:

```
void f3(char&);
void x3() {
      f3('c');          // illegal, 2.1 warns under +w
                        // illegal, 3.0 warns by default
}
```

Furthermore, since standard conversions were preferred to conversions requiring temporaries, the *Reference Manual* specified that the call to `f4()` below would be resolved to `f4(int)`. Instead, Release 2.0 resolved it to `f4(char&)`:

```
void f4(int);
void f4(char&);
void x4() {
      f4('c');      // illegal, 2.1 warns under +w
                    // illegal, 3.0 warns by default
}
```

Under the new rules, the calls to `f3()` and `f4()` are in error because a non-`const` reference cannot be initialized with a non-lvalue (see §8.4.3). However, Release 2.1 and 3.0.1 allow this behavior, with warnings, to provide the opportunity to migrate old code. Release 3.0.1 correctly warns by default in both case. Release 2.1 warns under `+w`.

Improved Operator Overloading (§13.5)

Operator overloading and the resolution of operator expressions has been more clearly specified for Release 3.0.1, notably in the area of choosing between user-defined operators and built-in operators using conversions to basic types. For instance, given the following class definition: class Foo { public: operator int(); int operator+(const Foo&,int); }; and an object of class `Foo`, `foo`, the expression `foo + 1` could be resolved two ways. It could be

resolved as `operator+(foo,1)` by calling the user-defined + operator, or as operator `int(foo) + 1` by using the built-in + operator on integers after applying the user-defined conversion to `int`.

For Release 3.0.1, the operator overloading algorithm has been updated to match the function matching algorithm. Therefore, argument matching is used to compare built-in operators to user-defined operators.

The only exceptions to this rule are operators which *must* be defined as members, i.e., `operator=()`, `operator[]`, `operator->()`, `operator()()`. For expressions involving these operators, the user-defined version of the operator is always preferred.

The effect of this clarification is that some expressions involving operators which used to call a user-defined operator will now be ambiguous. Other expressions which used to give an ambiguity error will now be resolved. For example,

```
class String {
public:
     String(char);
     friend Stringoperator+(String&,char); };
class MyClass {
public:
     operator int();
     friend int operator+(MyClass&,int);
     int operator[](unsigned int);
};
main()
{
     MyClass a;
     int i;


     i = a + 3;       // 1: used to call operator+(MyClass&,int);
                      // still does
     i = a + 3.2;     // 2: used to call operator+(MyClass&,int);
                      // now ambiguous
     i = a[3];        // 3: used to call operator[](unsigned int);
                      // still does
     i = 3 + a;       // 4: used to be ambiguous
                      // now calls built-in +
}
```

In call 1, Release 3.0.1 uses argument matching and chooses the user-defined operator. The best match on the first operand is the user-defined `operator+()`; the best matches on the second operand are both the user-defined `operator+()` and the built-in `operator+()` on integers. Thus, the intersection of best match functions is the user-defined `operator+()`.

Changing the right operand to a double makes call 2 ambiguous when using argument matching because the best match on the second operand will now be the built-in `operator+()` on doubles.

Call 3 still calls the user-defined `operator+()` because the user-defined version of `operator[]` is always preferred, since it must be defined as a member.

The last call (4) was ambiguous in pre-Release 3.0.1 versions of C++ because the call of the built-in `operator+()` on integers conflicted with `operator+(String&,char)`. Using argument matching, the call resolves to the built-in `operator+()` as the user would expect.

Miscellaneous Fixes & Enhancements

- Classes with destructors are now permitted in `||` and `&&` expressions.
- The limit on the size of inlines has been increased so that larger inline functions should now be laid down inline.
- The number of nested include files that `cfront` can handle has been made dynamic. The limit in Release 2.0/2.1 had been 127. Note that, of course, local `cpp`'s may vary in the limit they can process.
- Significant improvements have been made and extensive testing has been performed on the `+a1` (ANSI) option.
- Error messages for ambiguous function calls have been enhanced. The error message now lists the set of overloaded functions which were equivalently good.
- All known line numbering bugs are fixed.

Release 3.0.1 supports a return value optimization which may avoid the copying of potentially large data structures which are returned from functions.

For instance, given the following class definition and function declaration:

```
class T {
      ...
public:
      T(const T&);
};

T foo();
```

An object of class T may be initialized with the return value of foo() as follows:

```
T x = foo();
```

Such a function, `foo()`, will often have a definition something like the following:

```
T foo()
{
     T result;
     // do stuff to result
     return result;
}
```

This means that in order to do the above initialization, a copy will be done of `result` into `x`. If instead the function and the initialization had been written to look as follows:

```
void foo(T& result)
{
     // construct result
     // do stuff to result
     return;
}

T x;
foo(x);
```

the copy would be avoided altogether while achieving the same results.

Under certain conditions, `cfront` will now perform a transformation from the original, more natural, definition of `foo()` to the second definition automatically, thus avoiding the copy on the return.

This return value optimization is done under the following conditions:

- The function returns an object of type `T`, where `T` has a copy constructor, and
- The function creates a local variable of type `T`, say `result`, which is declared and returned at the top block of the function.
- The function does not return anything but `result`, from anywhere in the function between the declaration and return of `result`.

This optimization can eliminate the non-intuitive tricks that programmers often use to avoid copying of large objects on returns.

2.3 *Upgrading from Release 2.0 to Release 2.1*

Release 2.1 of the C++ Language System is source compatible with Release 2.0. That is, a legal C++ program that compiled and executed correctly with Release 2.0 will continue to compile and execute correctly with Release 2.1.

In addition, Release 2.1 is link compatible with Release 2.0. This means that libraries that were compiled using Release 2.0 do not need to be recompiled before linking with programs compiled with Release 2.1.

This section lists changes in Release 2.1. Most of these changes are bug fixes that have been made so that Release 2.1 more accurately reflects the definition of the C++ language given in the *Reference Manual.*

This section covers the following topics:

- *Header Files* — tells you about changes to the header files in Release 2.1
- *Changes to the* CC Command — tells you about changes in options to the CC command, macro name changes, and other changes in functionality
- *Language-Related Fixes* — tells you about fixes to the compiler that enforce language rules more accurately
- *Reference Manual* Changes — describes differences between the Release 2.0 *Reference Manual* and the Release 2.1 *Reference Manual.*
- *New Warning Messages* — lists warning messages that have been added for Release 2.1
- *Library Changes* — describes changes to the libraries supplied with Release 2.1

Recompilation of Release 2.0 Code Not Required

Code compiled using Release 2.0 does *not* need to be recompiled.

You might, however, want to recompile your old code using Release 2.1 anyway, as Release 2.1 enforces some language rules that were not enforced by Release 2.0. If you recompile your code, you will find out if it makes use of constructs that are illegal.

Header Files

Header File Bug Fixes

Bug fixes made to header files for Release 2.1 fall into several categories:

- Missing prototypes were added,
- Prototypes for functions specified by the ANSI C standard were updated to match the prototypes in the ANSI specification,
- Some headers that were missing for certain platforms have been added.

`stdlib.h` and `libc.h`

In Release 2.0, `stdlib.h` and `libc.h` were similar, but not identical. In Release 2.1, they are identical. `stdlib.h` is the ANSI C–specified header file used to declare many standard C library functions previously undeclared in C header files. `libc.h` is retained for compatibility with previous releases of the C++ Language System.

`curses.h` Proto-Headers Reorganized

Because of the great differences between various versions of `curses.h`, the proto-header for `curses.h` has been divided into three separate files: one for SVR 2 (`c proto-headers/curses.svr2`), one for SVR 3 (`c proto-headers/curses.svr3`), and one for all the other systems supported (`proto-headers/curses.h`).

In addition, the `curses.h` header for SVR 3 has been upgraded to SVR 3.2.

Changes to the `CC` Command

`a.out` File Permissions

Under Release 2.0 the `CC` command left the resulting `a.out` file with executable permission even if the `munch` or `patch` step of the compilation process failed. The Release 2.1 `CC` command does not make the `a.out` file executable if the `patch` or `munch` step of the process fails.

+L Option

The +L option had no effect in Release 2.0 because the compiler always generates source line information using the format `#line %d`. The +L option has therefore been removed from the CC man page for Release 2.1.

-Fc Option

The -F and -Fc options produce identical results in Release 2.0 and Release 2.1. They both run only the preprocessor and the compiler on the source files and send the generated C source code to the standard output. Therefore the -Fc option has been dropped as a separate option on the CC man page for Release 2.1, although it is still implemented.

Position-Independent Options

Options such as -Y, +a[01], -E, -F, -C, -P, -H, -S, -c, -I, -D, -U and -g are no longer position-dependent on the command line. Instead, they apply to all files specified on the command line. For example, under Release 2.1 the command:

```
CC foo.c -DDEBUG bar.c
```

defines the macro DEBUG for both foo.c and bar.c, whereas in Release 2.0 DEBUG was only defined for bar.c.

Not all options have been made position-independent, however. The +d, +p, and +w options are still position-dependent, as they were in Release 2.0. These options affect only those files named after the option is specified; the files named before the option are not affected. For example, the following command causes the +w option to be applied only to y.c, and not to x.c.

```
CC x.c +w y.c
```

The +e[01] options are also still position-dependent. Each +e option applies to all files listed before the next +e option is encountered. For example, in the case below +e0 is applied to the files x.c and y.c, whereas +e1 is applied to z.c:

```
CC +e0 x.c y.c +e1 z.c
```

The +a option specifies whether "Classic" C code or ANSI C-conforming code should be produced. Because the CC command invokes a single C compiler, it is assumed that only one setting of the +a option is appropriate. If multiple

+a[01] options are specified on the command line, the last option is the one actually used, and it is applied to all files. For example, the following command causes the +a1 option to be applied to x.c, y.c, and z.c.

```
CC x.c +a0 y.c +a1 z.c
```

Partial Compilation Options

If the options specified to the CC command contain a combination of the -P (run only the preprocessor step), -S (stop after creating the assembler input), and -c (compile but do not link) options, the option referring to the earliest stage of compilation is chosen and the others are ignored. For example, the following invocation causes the CC command to perform the preprocessing step *only* on the three files:

```
CC x.c -P y.c -S z.c
```

Virtual Table Optimization Improved

Release 2.1 provides the same virtual table strategy that was provided by Release 2.0.

Release 2.1 provides a further improvement on the treatment of virtual tables. Under Release 2.0, each virtual table had a companion pointer variable, which was used to hold housekeeping information necessary for the virtual table optimization. Under Release 2.1 these pointers are allocated in an array, rather than one per virtual table, so that only one symbol table entry is required in the generated object file. This change reduces the symbol table size (but not the runtime data size) of programs compiled with Release 2.1.

The new optimization is link compatible with Release 2.0.

More Debugging Information Generated Under the -g Option

Under Release 2.0, the -g option, which causes additional debugging information to be generated, was only passed to the underlying C compiler; it did not affect the behavior of the compiler itself. Under Release 2.1, however, the -g option also affects the behavior of cfront. If -g is specified, the compiler produces C code for *every* declaration in the compilation, rather than only for those declarations that are actually needed or used. This additional information allows for easier debugging, but it also increases the size of the object file because the symbol table is larger.

Warnings about Inline Functions Issued under the `+w` *Option*

Several customers have noted that Release 2.0 did not treat consistently inline functions that cannot be successfully inlined. Release 2.1 addresses this problem by providing more consistent information about whether inline functions are actually being inlined.

There are several cases:

- If an inline function is seen for which `cfront` cannot generate inline code, and `cfront` cannot recover from the error condition, a "not implemented" message is reported. (The "not implemented" messages are described in Appendix D of the Release 2.1 *Reference Manual.*)
- If an inline function is seen which cannot be inlined for some other reason (e.g., it is too long or it is a virtual function), and `cfront` can recover, the function will not be inlined and a warning message will be issued if the `+w` option is specified.
- If a *call* to an inline function is seen and, because of the characteristics of the call site, the particular call cannot be generated inline, a warning message will be issued if the `+w` option is specified.
- If the address of an inline function is taken, a warning message will be issued if the `+w` option is specified. Because the `inline` keyword is a "hint" to the compiler, and because the C++ Language System issues warnings unconditionally only about constructs that are almost certainly serious problems, warnings about inlines are issued only if the `+w` option is specified.

The following code illustrates the treatment of inlines:

```
inline in faint {return i;}

in g(int i) { return f(i); }

inline void h() {
      static int i = 5;
      // ...
}

struct S {
      virtual void f() {}
};
```

If you compile this code using `CC +w` you get the following output:

```
line 5: sorry, not implemented: cannot expand inline function with
static i
line 10: warning: virtual function S::f() cannot be inlined
line 12: warning: out-of-line copy of S::f() created
```

For more information about inline functions, see Chapter 8 of the *Selected Readings*.

Language-Related Fixes

This section describes bug fixes in Release 2.1 that may break some code that used to be accepted, but should never have been accepted. Section numbers (§) following a heading identify the section of the Release 2.1 *Reference Manual* that describes the correct behavior.

Implicit Conversions of Pointers to Members (§4.9)

Release 2.0 incorrectly permitted several kinds of implicit conversions involving pointers to members.

- Implicit conversions between pointers to members of unrelated types were permitted:

```
struct X { int i; };
struct Y { int i; };
int X::*pmXi = &Y::i;    // error
```

- Conversions from pointers to objects to pointers to members were also allowed:

```
struct Z { int i; };
int i;
int Z::*pmZi = &i;// error
```

- Finally, conversion from a pointer to member of a base class to a pointer to member of one of its derived classes was permitted:

```
struct B { int i; };
struct D : B { int i; };
int B::*pmBi = &D::i;// error
```

Release 2.1 correctly enforces these rules and reports an error in these cases.

Casts of Pointer Types (§5.5)

The *Reference Manual* states that a pointer may be explicitly converted to any integral type large enough to hold it. If the integral type is not large enough, the conversion is illegal. Release 2.1 enforces this rule; Release 2.0 did not.

```
char *p;
unsigned short us = (unsigned short) p;// error
unsigned short us1 = (unsigned short) (int) p;// ok
```

Better Enforcement of `const` (§7.2)

Release 2.0 did not always realize that a member of a `const` object is itself a `const`. For example, the assignment to `b.a.i` in the code below was permitted, even though `b` is `const` and therefore its members are also `const`.

```
struct A {
      int i;
};
struct B {
      A a;
      B();
};
void f() {
      const B b;
      b.a.i = 5; // error
}
```

Release 2.1 issues the following message:

```
line 10: error: assignment to member A::i of const B
```

Initialization of `const` Class Objects (§7.2)

The *Reference Manual* states that all `const` objects not explicitly declared to be `extern` must be initialized. Although Release 2.0 enforced this rule for built-in types, it did not require explicit initializations for `const` class objects, such as `a1` in the example below:

```
struct A { int a; };
struct B { B(); };
const A a1;          // error, no initializer
const A a2 = { 1 }; // ok, explicit initialization
const B b1;         // ok, implicit initialization by constructor
A a3;                // ok, non-const
```

Release 2.1 generates the following error for this code:

```
line 3: error: uninitialized const ::a1
```

Linkage Specifications (§7.6)

Release 2.0 did not enforce all the constraints on the use of linkage specifications. For example, it allowed a function declaration without a linkage specification to precede one with a linkage specification. This error is flagged by Release 2.1

```
int f();
extern "C" int f();// error
```

```
.line 2: error: inconsistent linkage specifications for f()
```

Local Variables in Default Arguments (§8.3, §10.5)

The *Reference Manual* forbids the use of local variables in default argument expressions. For example,

```
void f(int i) {
void g(int = i);
// ...
}
```

causes Release 2.1 to report the following error:

```
line 2: error: local i used as default argument
```

This error was not reported by Release 2.0.

Braced Initializers for Aggregates (§8.5)

The *Reference Manual* states that braced initializers may be used to initialize aggregates, which by definition cannot have private or protected members, constructors, base classes, or virtual functions. Release 2.0 did not enforce this rule for classes with private members, or for aggregate members that were not themselves aggregates. For example, Release 2.0 incorrectly allowed both initializations shown below.

```
class A {
     int a;
};
struct B {
      A obj;
};
A a = { 5 };      // error
B b = { 5 };      // error
```

Release 2.1 correctly generates errors for the initializations of `a` and `b`:

```
line 9: error: cannot initialize ::a with initializer list
line 10: error: cannot initialize ::b with initializer list
```

`const` Violations in `const` Member Functions (§9.4)

Release 2.0 did not consistently detect `const` violations in `const` member functions. For example, the following code is illegal because the value of `this`, which has type `const S *const`, is assigned to an object of type `S *const`. Because this code was accepted, illegal assignments to members within `const` member functions, such as the assignment to `i`, were not detected.

```
struct S {
      int i;
      void f() const {
            S *const p = this;// error
            p->i = 5;
      }
};
```

Release 2.1 correctly reports the following error for this code:

```
line 4: error: S::f() const: assignment of S::this (const struct S
*const) to S *const
```

`volatile` Member Functions Not Implemented (§9.4)

Release 2.1 issues a "not implemented" error message if a `volatile` member function is seen. Release 2.0 silently ignored the keyword `volatile` when applied to a member function.

Member Functions in Local Classes Must Be Defined Inline (§9.9)

When a class is defined within a function definition (that is, a local class), all member functions of the class must be defined within the class definition itself or not at all.

For example, the following code declares the function `f2()` but fails to define it:

```
void f() {
     struct Local {
           int f1() { return 0; }
           int f2(int);
     };
     Local var;
}
```

Release 2.0 quietly accepted the above code. Release 2.1, however, issues the following warning:

```
line 6: warning: f2() must be defined inline within local class Local
```

Protection Violations of Anonymous Union Members (§11.1)

Release 2.0 did not enforce access protection for members that are anonymous unions. For example, the following code was silently accepted:

```
class S {
      union { int i; double d; };
};
void f() {
      S s;
      s.i = 5; // error
};
```

Release 2.1 correctly reports an error for the assignment to `s.i` because `i` is declared in the private part of `S`.

Friend Declarations Cannot Be Class Definitions (§11.5)

The syntax for declaring a class to be a friend of another class allows the use of an *elaborated-type-specifier*, but not a complete class definition, in the declaration. Therefore, the first friend declaration in the example below is legal, but the second is not.

```
class C {
      friend struct A; // ok
      friend struct B { int f(); }; // error
};
```

Release 2.0 did not recognize the error in the friend declaration for `B`, but Release 2.1 issues the following error message:

```
line 3: error: friend struct B {...}
```

Access to Protected Members (§11.6)

The *Reference Manual* states that a derived class may refer to a protected member of a base class *only* if the reference is through a pointer to, reference to, or object of the derived class. For example, in the code below, although `class D` is derived from `class B`, `D::f()` cannot call the protected function `B::g()` through a `B` pointer. The same rules apply to constructors, making the calls to `B::B()` in `D::f()` illegal.

```
class B {
      B(int);
      void g(int);
protected:
      B();
      void g();
};
class D : public B { void f(); };
void D::f() {
      B b;              // error
      B* bp = new B;    // error
      bp->g();          // error
}
```

In general, Release 2.0 reported the protection violations in code such as this. In some cases, however, no errors were reported. Such cases generally involved overloaded functions, one of which was protected, as shown in the above example.

Release 2.1 correctly generates the following messages for this code:

```
line 11: error: D::f() cannot access B::B(): protected member
line 12: error: D::f() cannot access B::B(): protected member
line 13: error: D::f() cannot access B::g(): protected member
```

Redundant Initializers (§12.7)

The following code was accepted by Release 2.0 but is incorrect because it specifies two initializers for the same object. Release 2.1 reports an error.

```
struct Point { Point(int, int); };
void f() {
      Point p(1, 2) = Point(3, 4);// error
}
```

Illegal Function Overloading (§13.1)

The *Reference Manual* states that functions with parameter types that differ only with respect to `const` or `volatile` may not have the same name. Release 2.0 did not enforce this rule consistently and accepted code such as the following:

```
void f(int *);
void f(int *const);// error
```

Release 2.1, however, correctly reports an error for the second declaration of `f()`.

```
line 2: error: the overloading mechanism cannot tell a void (int *)
from a void (int *const)
```

"Intersection Rule" Applied to Function Matching (§13.3)

Release 2.0 did not fully implement the "intersection rule" for function matching described in §13.2 of the *Reference Manual*. For example, the following code was accepted and a call to `f(double,double)` was generated.

```
double f(double, double);
double f(float, float);
double d = f(double(1.0),float(1.0));// ambiguous
```

According to §13.2, however, this call is ambiguous. If you look for possible matches, parameter by parameter, you see that the set of best matches for the first parameter has only one element, `f(double,double)`, and the set of best matches for the second parameter also has only one element, `f(float,float)`. The intersection of these sets is empty, so the call is ambiguous.

For this example, Release 2.1 correctly issues the following message:

```
line 3: error: ambiguous call of f(); double (double, double) and
double (float, float)
```

This change in behavior may affect class libraries that provide functions that overload system functions. For example, suppose you define a type `String` and then overload the system function `read()` to handle objects of type `String`:

```
struct String {
String(char*);
// ...
};
int read(int, String&, int);
```

However, you do not notice that the last parameter of the system `read()` function is an `unsigned` rather than an `int`:

```
int read(int, void*, unsigned);// system 'read()'
```

Because Release 2.0 did not correctly implement the intersection rule, calls to the library's `read()` were considered unambiguous. Under Release 2.1 they are ambiguous because the intersection rule is strictly applied:

```
void g(int fd, char* cp) {
      (void) read(fd, cp, 3);// ambiguous
}
```

The point here, especially for library writers, is to be careful when overloading system functions. The types of the parameters that are intended to be the same should match exactly.

Restrictions on Overloaded Operators (§13.5)

The *Reference Manual* places a number of restrictions on the ways in which operators can be overloaded. For example, `operator=()` must be a non-static member function. Release 2.1 enforces these rules more strictly than Release 2.0 did.

```
struct S {
static operator=(int);// error
};
```

Release 2.1 reports the following error for the above example:

```
line 2: error: S::operator=() cannot be a static member function
```

Reference Manual **Changes**

Release 2.1 provided a new, revised *Reference Manual*, which incorporated hundreds of customer comments on the draft *Reference Manual* distributed with Release 2.0. The Release 2.1 *Reference Manual* clarified the wording and intent of the language definition, corrected errors, and removed inconsistencies. In a very few cases, the language rules were deliberately changed, in response to feedback from programmers using C++. The revised *Reference Manual* was submitted to the American National Standards Institute (ANSI) and has been accepted as the basis for standardizing the C++ language. An annotated version of the *Reference Manual*, entitled *The Annotated C++ Reference Manual*, was published in early 1990 by Addison-Wesley.

This section lists the changes in the Release 2.1 *Reference Manual*, ordered by section of the *Reference Manual*. To help you determine quickly which changes might impact your code, each change has been classified into one of the following categories:

- *Extension*, which is implemented in Release 2.1
- *Restriction*, also implemented in Release 2.1
- *Clarification*, which makes a language rule more explicit and which does not affect the behavior of the C++ Language System
- *Change*, for which no corresponding change has yet been made to the C++ Language System

---

Note – Release 2.1 will continue to compile successfully every legal C++ program that compiled under Release 2.0. As usual, you will get a warning message if you use a construct that is no longer legal, but your program will still compile just as it did under Release 2.0. If your program compiles without any anachronism warnings, then it will work the same way when the new rules are completely phased in and the old rules are completely phased out. Remember that some anachronism warnings appear only if `+w` is specified.

New Keyword `try` (§2.5, restriction)

There is a new keyword, `try`, for exception handling. Although Release 2.1 does not implement exception handling, a warning message is issued if an identifier named `try` is encountered.

```
int try;
line 1: warning: try is a future reserved keyword
```

Note – Release 3.0.1 recognizes the full exception handling syntax, but issues a "sorry, not implemented" message.

One Definition of an Inline Member Function (§3.4, change)

According to the Release 2.1 *Reference Manual*, an inline member function must have exactly one definition in a program. In other words, an inline member function cannot legally have different definitions in different files. Previously, this restriction was not explicitly stated.

This rule might be easily enforced in a C++ environment where a library manager keeps track of all definitions in a program, but the C++ Language System does not enforce this rule.

Character Types (§3.7, clarification)

The Release 2.1 *Reference Manual* states that the types `char`, `unsigned char`, and `signed char` are three distinct types. This corrects a misstatement in the previous *Reference Manual* and conforms with the ANSI C standard.

Because the C++ Language System ignores the keyword `signed`, Release 2.1 provides two character types: `char` and `unsigned char`.

Qualified Name Syntax for Nested Types (§5.2, §9.8, extension)

The Release 2.1 *Reference Manual* extends the qualified name syntax to apply to type names as well as class members. This new syntax allows a nested type to be named outside the class in which it is defined.

For example, to refer to the enumeration type `E` outside the definition of `Outer`, the syntax `Outer::E` should be used, as shown below.

```
struct Outer {
      enum E { e };// nested type
};

Outer::E var1;// use of a nested type
```

To provide compatibility with Release 2.0, Release 2.1 also allows you to refer to a nested type name without qualification, as in the following declaration:

```
E var2;
```

Release 2.1 issues a warning message, however, for this use:

```
line 1: warning: use Outer:: to access nested enum type E
(anachronism)
```

The qualified name syntax is recursive, but Release 2.1 does not implement qualified names with more than two identifiers:

```
struct S1 {
      struct S2 {
           typedef int T;
      };
};

S1::S2::T var3;            // legal, sorry in 2.1
```

For this code, the following message is issued:

```
line 7: not implemented: class names do not nest, use typedef x::y
y_in_x
```

Note – True nested types are implemented in Release 3.0.1, and the transition model supplied in Release 2.1 is no longer supported.

Class Arguments to `f(...)` (§5.3, extension)

The Release 2.0 *Reference Manual* specified that it was illegal to pass an object of a class with a constructor to a function with an ellipsis formal parameter. This restriction is lifted in the Release 2.1 *Reference Manual* and the new behavior is implemented in Release 2.1. The following code, which produced an error under Release 2.0, compiles without complaint under Release 2.1. The copy constructor is *not* invoked to pass the argument. Instead, a bit-wise copy is done.

```
struct S {
      S();
      S(const S&);
};
void f(...);

void g(S s) {
      f(s);                // legal, accepted by 2.1
}
```

Explicit Type Conversions with Empty Initializers (§5.4, change)

The 2.1 *Reference Manual* allows you to specify an explicit type conversion with an empty initializer, as in the following examples:

```
int i = int();

struct Empty {};
Empty e = Empty();
```

Release 2.1 does not implement this capability and reports an error instead.

```
line 1: error: value missing in conversion to int
line 4: error: cannot make a Empty
```

Note – Release 3.0.1 implements this capability.

Size of a Function (§5.4, restriction)

In C++, as in ANSI C, you are allowed to apply the `sizeof` operator to a pointer to a function but not to the function itself. For example, this code is legal:

```
void f();
int i = sizeof(&f);
```

but this is not:

```
int j = sizeof(f);
```

Release 2.1 enforces this restriction.

Access Protection for `operator new()` (§5.4, restriction)

Release 2.0 did not check access protection for calls to class-specific `operator new()`. The Release 2.1 *Reference Manual* explicitly extends access protection to calls to class-specific `operator new()`, and Release 2.1 implements this behavior. For example, the following code compiled without error under Release 2.0, but produces an error message under Release 2.1.

```
#include <stddef.h>
class C {
     void* operator new(size_t);
     void operator delete(void*);
public:
     C();
};

void f() {
      C *cp = new C;// illegal, error in 2.1
}
```

Release 2.1 issues the following diagnostic for this code:

```
line 8: error: f() cannot access C::operator new(): private member
```

Empty Initializers for `operator new()` (§5.4, clarification)

The Release 2.1 *Reference Manual* explicitly allows the initialization expression in an allocation expression to be empty, as in the following examples:

```
double* dp = new double();
struct Complex {
Complex();
// ...
};
Complex* cp = new Complex();
```

For a built-in type, this means that an object with an undefined value is created. For a class type, this means that the default constructor is called. If there is more than one default constructor, an error is reported because the call is ambiguous. If there is no default constructor, an object with an undefined value is created.

Both Releases 2.0 and 2.1 implement this behavior correctly.

Deleting an Array (§5.4, extension)

It used to be necessary to specify the number of elements when deleting an array. For example, you were required to specify the expression `10` when deleting the array pointed to by `p` in the following code:

```
struct S { S(); ~S(); };
void f1() {
      S *p = new S[10];
      // ...
      delete [10] p;
}
```

With Release 2.1 this is no longer necessary, and the following code is now accepted:

```
void f2() {
      S *p = new S[10];
      // ...
      delete [] p;// no size necessary
}
```

Use of the old syntax is considered an anachronism, and Release 2.1 issues the following diagnostic if the `+w` option is specified to the `CC` command:

```
line 4: warning: v in 'delete[v]' is redundant; use 'delete[]'
instead (anachronism)
```

This capability frees the programmer from having to keep track of array sizes. It also prevents subtle problems caused by discrepancies between the number of allocated elements and the number of deleted elements.

Note – Release 3.0.1 issues an unconditional warning if this syntax is detected.

When an array is created using the placement version of `operator new`, destruction and deletion of that array are the user's responsibility. For example:

```
class T {
      T();
      ~T();
      // ...
};

T*
create_T_array_in_buffer(void* buff, int n)
{
return new (buff) T[n];
}

void foo()
{
      pv = malloc(sizeof(T)*5);
      T* pT = create_T_array_in_buffer(pv, 5);
      delete [] pT; // does not work!!!
}
```

Here are some approaches the user can take to this problem:

```
delete [5] pT;
```

This (anachronistic) syntax will run the destructor on the objects in the array and free the storage using the global operator `delete`. Possibly this syntax should be resurrected.

```
T* ppT = pT + 5;
while (pT <= --ppT)
      ppT->T::~T();
```

This loop destroys the objects in the array but does not free the storage, appropriate in case the storage is managed by specialized code.

Type Definitions in Casts (§5.5, clarification)

The Release 2.1 *Reference Manual* clearly states that it is illegal to define a type in a cast. For example, the following declaration is illegal and is rejected by the C++ Language System:

```
enum E { e1 = (enum { z = 10 } ) 3, e2 };// error in 2.0 and 2.1
```

Declarations in `for` Initializers (§6.6, §6.8, clarification)

The Release 2.0 *Reference Manual* stated that a `for` statement containing a declaration in its *for-init-statement* was not allowed to be the statement after an `if`, `else`, `switch`, `while`, `do`, or `for`. In other words, this code was illegal:

```
void f(int i) {
      if (i)
           for (int j = i; j; j--)// error
                ;
}
```

This restriction was an error not enforced by the Release 2.0 implementation, and the Release 2.1 *Reference Manual* omits it.

The Release 2.1 *Reference Manual*, however, does specify a related restriction: "An `auto` variable constructed under a condition is destroyed under that condition and cannot be accessed outside that condition."

Here is an example:

```
int g(int i) {
      if (i)
           for (int j = 5; j; j--)
                ;
      return j;// error
}
```

In the above code, `j` cannot be accessed at the point of the `return` statement because the `return` statement is outside the body of the `if` statement. According to the Release 2.1 *Reference Manual*, an error should be reported, but Release 2.1 quietly accepts this code.

Note – Release 3.0.1 correctly reports the error.

Another example:

```
struct S {
      S(int);
      ~S();
      operator int();
      S& operator--();
};
int h(int i) {
      if (i)
           for (S s = 5; s; s--)
                ;
      return s;// error
}
```

The destructor for `s` is invoked at the end of the `if` statement. Release 2.1 (correctly) issues the error message

```
line 11: error: s undefined
```

at the `return` statement.

Global Inline Functions Are Static (§7.2, §7.2, §3.4, change)

The Release 2.0 *Reference Manual* allowed a non-member inline function to have external linkage. The Release 2.1 *Reference Manual* specifies, however, that a name of global scope that is declared `inline` is local to its file.

Release 2.1 does not conform to these rules. For example, the following code is accepted by Releases 2.0 and 2.1: `f()` is treated as a `static` function, and a `static` definition of `f()` is laid down.

```
extern int f(int);
inline int f(int i) { return i; }// error, not reported
int i = f(0);
int (*pf)(int) = &f;
```

Instead, the C++ Language System should report an error that `f()` cannot be redeclared as `inline` after being declared `extern`.

Use of `typedef` Name as Synonym for a Class Name (§7.2, clarification)

The Release 2.0 *Reference Manual* was not explicit about where a `typedef` name could be used in place of a class name. The Release 2.1 *Reference Manual* clarifies this: "The synonym may not be used after a `class`, `struct`, or `union` prefix and not in the names for constructors and destructors within the class declaration itself." These restrictions have not been implemented by Release 2.1.

```
struct S {
S();
~S();
};
typedef struct S T;
S a = T(); // legal, accepted by 2.0 and 2.1
struct T *p; // illegal, but accepted by 2.0 and 2.1
class C;
typedef class C U;
struct U {}; // illegal, but accepted by 2.0 and 2.1
```

Because typedef names cannot be used in the names of constructors, both Release 2.0 and 2.1 treat the use of a typedef name in a member function declaration as introducing an ordinary member function of that name, not a constructor. Since this is likely to be an error, the C++ Language System should, but does not, issue a warning. However, both Release 2.0 and 2.1 correctly reject the use of a typedef name in a destructor:

```
typedef struct X Y;
struct X {
      X();     // constructor
      Y(int); // illegal, accepted by 2.0 and 2.1
      ~Y();    // illegal, detected by 2.0 and 2.1
};
```

Scope of a Nested Enumeration (§7.3, §9.8, extension)

In conjunction with the introduction of nested types, the name of an enumeration type declared within a class declaration is local to the class. This marks a change from the Release 2.0 semantics. As a result of this change, the scope of an enumerator declared within a class is the same as the scope of its enumeration type.

```
struct S {
       enum E { e1, e2 };
       // ...
};
S::E var = S::e1;// 'E' and 'e1' have the same scope
```

`const` Functions (§8.3, restriction)

The Release 2.1 *Reference Manual* restricts the use of the `const` and `volatile` qualifiers to non-`static` member functions. Release 2.1 implements this restriction. Release 2.0 accepted the declarations of `g()` and `x()` below, whereas Release 2.1 correctly rejects them:

```
class C {
      int f() const;           // legal
      static int g() const;     // illegal, error in 2.1
};

void x() const;                // illegal, error in 2.1
```

Default Arguments Illegal for Overloaded Operators (§8.3, §13.5, restriction)

The Release 2.1 *Reference Manual* explicitly states that default arguments are illegal for user-defined operators. Release 2.1 implements this rule. The code below was accepted by Release 2.0 but is rejected by Release 2.1.

```
struct S {
      friend int operator+(S, int = 0);// illegal, error in 2.1
      // ...
};
```

Scope of a Class Member's Initializer (§8.5, clarification)

The Release 2.1 *Reference Manual* states explicitly that an initializer for a static member is in the scope of the member's class. This rule was not explicitly given in the previous *Reference Manual.*

Release 2.1 does not apply this rule consistently. For example, in

```
const int a = 5;

struct X {
     static int a;
     static int b;
};

int X::a = 1;
int X::b = a;
```

the correct behavior is implemented: `X::b` is initialized with `X::a`.

However, default arguments for member functions are not resolved within the scope of the class. In the following code,

```
const int y = 2;

struct Y {
     static int y;
     static int f(int);
};

int Y::f(int i = y) { return i; }
```

Release 2.1 incorrectly determines that the default argument for `Y::f()` is global `y`, not `Y::y`.

Note – Release 3.0.1 correctly resolves the argument.

Reference Initializers (§8.5, restriction)

The Release 2.0 *Reference Manual* allowed a reference to be initialized with a temporary, as in the following declaration:

```
int& r = 5;
```

However, the Release 2.1 *Reference Manual* has tightened the rules for reference initializations so that only `const` references may legally be initialized with non-lvalues. This means that, instead of the previous declaration, you must use the following:

```
const int& cr = 5;
```

The Release 2.0 C++ Language System already treated temporary initializers for non-`const` reference initializations at global scope as errors, although it allowed them at local scope. To provide a smooth transition to the more restrictive rules, Release 2.1 issues an anachronism warning, under control of the `+w` option, for non-`const` reference initializations that were accepted by Release 2.0 but are now illegal.

Here are some examples:

```
int& r1 = 5;              // illegal, error in 2.0 and 2.1

struct A { A(int); ~A(); };
A& a1 = 5;                // illegal, sorry in 2.0, error in 2.1
const A& a2 = 5;          // legal, sorry in 2.0, bad code in 2.1

int& f1();
int& r2 = f1();           // ok, 'f()' returns an lvalue

const int& r3 = 5;        // ok, 'r3' is 'const int&'

int f2(int&);
int j = f2(5);            // illegal, error in 2.0 and 2.1

void x() {
      int& r1 = 0;       // illegal, 2.1 warns under +w
      A& a1 = 5;         // illegal, 2.1 warns under +w
      const A& a2 = 5;   // legal, accepted by 2.0 and 2.1
      int j = f2(5);     // illegal, 2.0 and 2.1 warn under +w
}

struct S1 {};
struct S2 {
operator S1();
};

void f3(S1&);
void y(S2 s2) {
f3(s2);                 // illegal, 2.0 and 2.1 warn under +w
}
```

Note – Release 3.0.1 issues an unconditional warning, or an error if the `+p` option is in effect.

The anachronism warnings turn into errors if the `+p` option is specified to the `CC` command.

Reuse of a Class Name by its Members (§9.3, clarification)

The Release 2.1 *Reference Manual* limits the ways in which a class name can be reused by members of the class. The rule is that a static data member, enumerator, member of an anonymous union, or nested type may not have the same name as its class.

Release 2.1 does not enforce these restrictions completely. An error is reported if an enumerator or nested type has the same name as its enclosing class, but a static data member or member of an anonymous union are not caught.

```
struct S1 {
static int S1;              // illegal, no error in 2.0 or 2.1
};
struct S2 {
union { int i; float S2; };// illegal, no error in 2.0 or 2.1
};
```

Static Data Members of Local Classes (§9.5, change)

The Release 2.1 *Reference Manual* states that static data members are not allowed for local classes. Previously, a local class could have a static data member only if no explicit initialization was required.

Release 2.1 does not enforce the new restriction properly. If a static data member of a local class is declared but never used, a warning is reported but the program links successfully.

```
int main() {
      struct S {
            static int i;
      };
      // ...
      return 0;
}
```

```
line 2: warning: static member S::i in local class S (anachronism)
```

Note – Release 3.0.1 enforces this restriction, and correctly reports an error.

If the static data member is used, the program usually cannot be linked.

```
int main() {
      struct S {
            static int i;
      };
      S::i = 5;
      // ...
      return 0;
}
```

When the above code is compiled and linked, the following messages are reported on UNIX System V. They indicate that the static member `S::i` was declared but never defined:

```
CC x.c:
line 2: warning: static member S::s in local class S
cc -Wl,-L/c++/cfront/cycle16 x.c -lC
undefined                    first referenced
symbol                            in file
S__main__Fv__L1::s                /usr/tmp/CC.28949/x.o
ld fatal: Symbol referencing errors. No output written to a.out
```

No Virtual Functions in Unions (§9.6, clarification)

Because a union cannot be used as a base class, it makes no sense for member functions of unions to be declared `virtual`. The Release 2.1 *Reference Manual* states this restriction explicitly, and Release 2.1 implements it.

```
union U {
int i;
double d;
virtual int f();// error
};
```

Release 2.1 reports the following error for this code:

```
line 3: error: f(): cannot declare virtual function within union
```

Introduction of True Nested Types (§9.8, extension)

The Release 2.1 *Reference Manual* introduces true nested types. In previous versions of the C++ language, as well as in C, nested classes are treated as a lexical convenience; they are "hoisted" to the scope of the enclosing class. With Release 2.1, however, all names declared within a class definition are local to the class and are not hoisted. The new rules provide greater consistency, improved modularity, and more intuitive behavior. In addition, they remove some of the anomalies that previously occurred with nested local classes.

To avoid breaking code that worked under Release 2.0, Release 2.1 implements a transition model for nested types, which is designed to preserve the behavior of existing programs while allowing a smooth transition to the new semantics. The old Release 2.0 behavior is now considered anachronistic.

Note – True nested types are implemented in Release 3.0.1, and the transition model supplied in Release 2.1 is no longer supported.

Briefly, the transition model consists of three rules:

- Programs that are legal under the old rules and mean something else under the new rules (legal or illegal) continue to follow the old rules, and a warning is issued. For example, the use of the nested type `E` below is illegal under the new rules, but because it was legal under Release 2.0, Release 2.1 issues a warning rather than an error.

```
class X {
      enum E { };
};
E e; // legal in 2.0, warning in 2.1
```

Release 2.1 issues the following warning for the above declaration of `e`:

Warning – Use `X::` to access nested enum type E (anachronism)

If the `+p` option (which disallows anachronistic constructs) to the `CC` command is specified, the anachronism warning turns into an error.

Here is an example of code that is legal under both old and new rules, but means different things:

```
extern int i;
struct S {
      static int i;
      struct Embedded {
            int f() { return i; }
      };
};
```

Under Release 2.0, `Embedded::f()` returned global `::i`, whereas under the new nested types rules, it should return `S::i`. In this case Release 2.1 issues a warning

```
line 6: warning: i , accessed within nested class Embedded, is
visible both globally
and within enclosing class S -- using ::i (anachronism)
```

and preserves the old behavior.

The `+p` option has no effect on this example; the warning does *not* turn into an error.

- Programs that are legal and mean the same thing under both sets of rules behave the same.
- Programs that are legal under the new rules and illegal under the old rules follow the new rules. For example, the new qualified name syntax was illegal under Release 2.0, but is legal under Release 2.1.

  ```
  X::E xe; // syntax error in 2.0, legal in 2.1
  ```

There is one case that causes difficulty for the transition model. Consider the following program, which is illegal under the old rules because the class `Nested` is defined twice:

```
struct S {
      class Nested {};
};

void f(Nested);

struct T {
       class Nested {}; // old rules in effect; illegal in 2.0
                        // and 2.1
};
```

This program fails under the transition model for a subtle reason. When the compiler sees the declaration of `f()`, it does not know whether `Nested` should be treated under the old or the new rules. It has to know so that it can decide how to encode the function name in the generated C code. For compatibility, it must assume the old rules. Thus when it sees the second definition of `Nested`, it reports an error.

To allow this program to compile, you must do something early on to force the program to be considered unquestionably illegal under the old rules. The easiest way to do this is to define a global class with the same name as the nested class *before* the nested class definition. In the example below, `Inner` is the nested type that is defined within two global classes and thus requires a dummy global definition:

```
struct Inner {};     // dummy class; tells the compiler
                     // to use the new nested types semantics
class C {
      class Inner { /* ... */ };// legal in 2.1, error in 2.0
};
void g(C::Inner) {}              // legal in 2.1, error in 2.0

class D {
      class Inner { /* ... */ };// legal in 2.1, error in 2.0
};
void g(D::Inner) {}              // legal in 2.1, error in 2.0
```

The above code compiles and links properly.

To preserve link compatibility with libraries compiled under Release 2.0, you should *not* force your programs to use the new rules, as is done with `Inner` in the example above. If the new rules are applied, then function names are encoded differently, and new code will not link with old libraries.

Nested Local Types (§9.8, §9.9, extension)

The transition model for nested types guarantees that code that is legal under both the old and new rules but that changes meaning under the new rules preserves its former meaning. In this situation, Release 2.1 issues an anachronism warning.

Note – Full nested types are implemented in Release 3.0.1, and the transition model supplied in Release 2.1 is no longer supported.

Here is an example that involves nested local types:

```
struct Nested { int i; };
typedef int T;
enum E { e };

void f() {
       struct Local {
              struct Nested { int i, j; };
              typedef double T;
              enum E { e };
       };
       Nested n1;
       T t1;
       E e1;
}
```

In this example, `n1` had type `Local::Nested` under Release 2.0 because the declaration of `Local::Nested` was exported into the scope of `f()`. Similarly, `t1` had type `double`. Release 2.0 incorrectly reported an error for the declaration of `E` within `Local`, so `e1` was also reported as an illegal declaration.

Release 2.1 preserves this behavior (except for the bogus error) and issues the following messages:

```
line 11: warning: Nested occurs at global and nested local class
         scope; using class type Local::Nested
line 12: warning: T occurs at global and nested local class scope;
         using typedef Local::T
line 13: warning: E occurs at global and nested local class scope;
         using enum type Local::E
```

Under the +p option to the CC command, the behavior does not change: `Local::Nested`, `Local::T`, and `Local::E` are still used. You are encouraged, however, to change your declarations to

```
Local::Nested n1;
Local::T t1;
Local::E e1;
```

to ensure that your code continues to have the same meaning after nested types are fully implemented.

Protected Derivation (§10.1, change)

The Release 2.0 *Reference Manual* explicitly disallowed the use of `protected` as an access specifier for a base class. The Release 2.1 *Reference Manual* lifts this restriction. However, Release 2.1 does not implement the new behavior.

```
struct B {};
struct D : protected B {};// legal, but rejected by 2.1
```

Note – Release 3.0.1 correctly implements the new behavior.

Extension of Dominance Rule to Objects and Enumerators (§10.2, change)

The Release 2.0 *Reference Manual* restricted the concept of dominance to apply only to functions. That is, dominance was used only when disambiguating function names in an inheritance hierarchy involving virtual base classes. With the Release 2.1 *Reference Manual*, the dominance concept is extended to data members and enumerators. However, Release 2.1 does not implement the new semantics. In the following example, Release 2.1 incorrectly considers the use of `x` to be ambiguous, even though `B::x` dominates `V::x`.

```
struct V { void f(); int x; };
struct B : public virtual V { void f(); int x; };
struct C : public virtual V {};

struct D : public B, public C { void g(); };

void D::g() {
      x++; // legal, but rejected by 2.0 and 2.1
      f(); // legal, accepted by both 2.0 and 2.1
}
```

Note – The extension of dominance to objects is implemented in Release 3.0.1.

Inheritance of Pure Virtual Functions (§10.4, extension)

The Release 2.0 *Reference Manual* required that a derived class define or declare pure every pure virtual function in its immediate base. This restriction is lifted in the Release 2.1 *Reference Manual*; pure virtual functions are now inherited as pure virtual functions. The new behavior is implemented in Release 2.1.

For example, the following code is legal under Release 2.1, but produced an error under Release 2.0:

```
struct A {                  // abstract class
      virtual void f() = 0;
};

struct A2 : public A {};
```

Although `f()` is not redeclared as pure virtual in `A2`, Release 2.1 (but not Release 2.0) considers `A2` to be an abstract class because `A::f()` is inherited as pure virtual.

Access Specifiers in Unions (§11.1, change)

The Release 2.1 *Reference Manual* allows access specifiers in unions. Formerly, these were forbidden.

```
union U {
public:          // legal
      U();
      int i;
private:         // legal
      double d;
protected:       // legal
      float f;
};

U u;
float f = u.f; // protection violation
```

Release 2.1 accepts the definition of `U` shown above but does not report the protection violation.

Note – Release 3.0.1 flags the protection violation.

Access to Static Members of Private Base Classes (§11.3, change)

The Release 2.1 *Reference Manual* states that a private derivation of a base class does not restrict access to the static members of the base class. Without this rule, a member function would have less access to a base class's static members than a global function.

Release 2.1 does not implement this rule consistently. For access to a static member of an immediate base class, some illegal accesses are not reported:

```
struct B {
static void f();
};

struct D : private B {}
struct E : private D {
      void g() {
            f();       // illegal, not reported by 2.0 or 2.1
            this->f();// illegal, not reported by 2.0 or 2.1
            B::f();    // legal, rejected by 2.0 and 2.1
      }
};
```

In the above code, the calls `f()` and `this->f()` are illegal because they refer to `f()` via the `this` pointer, and thus the access protection for `private` members is applied. The call `B::f()` is legal because it refers to `f()` directly, just as a global function could refer to `B::f()`.

Note – Release 3.0.1 implements the rule consistently.

If multi-level derivation is involved, both Releases 2.0 and 2.1 are overly conservative; they report an error for `X::f()` even though it is legal.

```
struct X {
        static void f();
};
struct Y : private X {};
struct Z : public Y {
        void g() {
                f();          // illegal, error in 2.0 and 2.1
                this->f();    // illegal, error in 2.0 and 2.1
                X::f();       // legal, error in 2.0 and 2.1
        }
};
```

Access Declarations (§11.4, clarification)

The Release 2.1 *Reference Manual* explicitly imposes the following restriction on access declarations: an access declaration may not adjust the access to a base class member if the derived class also defines a member of the same name.

This rule is implemented by both Releases 2.0 and 2.1:

```
struct B {
      int i;
};

struct D : private B {
      B::i;
      int i;          // error
};
```

Linkage of Friend Functions (§11.5, restriction)

The Release 2.1 *Reference Manual* specifies that the default linkage for `friend` functions is `extern`. For example,

```
static f();

struct S {
      friend f();// ok, internal linkage
      friend g();// 'g()' has external linkage
};

static g();// illegal, error in 2.0 and 2.1
```

Release 2.1 warns about static friend functions such as f() in the example above because, although legal, these could in principle be used to subvert the protection system. Release 2.1 issues the following messages for the example above:

```
line 3: warning: static f() declared friend to class S
line 8: error: g() declared as both static and extern
```

Friendship Is Not Inherited (§11.5, clarification)

The Release 2.0 *Reference Manual* incorrectly stated that friendship is inherited. The Release 2.1 *Reference Manual* corrects this mistake. In Release 2.1, as in previous releases of the C++ Language System, friendship is *not* inherited.

Friendship Applies to Non-Functions (§11.5, clarification)

The Release 2.1 *Reference Manual* makes it explicit that class friendship extends to all members of the class — not just to functions. Release 2.0 and Release 2.1 both implement this behavior. For example,

```
class X {
      enum { e = 100 };
      friend class Y;
};

class Y {
      int arr[X::e];      // legal, accepted by 2.0 and 2.1
};
class Z {
      int arr[X::e];      // error, 'X::e' is private
};
```

Scope of Friend Functions (§11.5, §9.8, change)

The Release 2.1 *Reference Manual* states that a `friend` function defined within a class declaration is in the lexical scope of that class, just like a member function.

In general, Release 2.1 does not implement this rule. Consider the following example:

```
extern int s;
extern int e;

struct S {
      static int s;
      enum { e = 5 };
      friend f() { return e; }      // which 'e'?
      friend void g(int = s) { };   // which 's'?
};
```

According to the Release 2.1 *Reference Manual*, `f()` returns `S::e` and the default argument for `g()` is `S::s`. Instead, both Release 2.0 and 2.1 incorrectly resolve these names to `::e` and `::s` respectively.

Note – Release 3.0.1 resolves these names correctly.

If the declaration of a `friend` function within a class declaration uses a nested type, however, the nested type name is resolved according to the new semantics.

```
typedef void* T;
struct X {
      typedef int T;
      friend T h(T t);
};
```

In the above example, Release 2.1 treats `h()` as having type `int(int)`, not `void*(void*)`.

Default Constructors (§12.2, change)

The Release 2.0 *Reference Manual* explicitly stated that a default constructor is a constructor with no formal parameters, thereby excluding constructors that can be called with no arguments by virtue of having default arguments. The Release 2.1 *Reference Manual* lifts this restriction; the constructor in the example below is now considered a default constructor.

```
struct S {
      S(int = 0);
};
```

Release 2.1 does not conform to this rule. Instead, it adheres to the old definition of default constructor. Here are some examples:

```
S s1[2];          // legal, sorry in 2.0, error in 2.1
S s2[2] = { 1 };  // legal, sorry in 2.0, sorry in 2.1

struct X {
      S s[2];// legal, sorry in 2.0, error in 2.1
};

void f() {
     S* p = new S[2];// legal, error in 2.0 and 2.1
}
```

Note – Release 3.0.1 correctly conforms to this rule.

Constructor and Destructor Declarations (§12.2, §12.5, §9.4, clarification)

The Release 2.1 *Reference Manual* specifies that constructors and destructors cannot be declared `const`, `volatile`, or `static`. Release 2.1 correctly reports an error for constructors and destructors that are declared `static`, but it

incorrectly allows constructors and destructors to be declared `const`. Release 2.1 does not implement `volatile` member functions at all; these are rejected with a "not implemented" message.

```
struct S {
      static S();          // illegal, error in 2.0 and 2.1
      static ~S();         // illegal, error in 2.0 and 2.1
};

struct T {
      T() const;           // illegal, but accepted by 2.1
      ~T() const;          // illegal, but accepted by 2.1
      T(char*) volatile;   // illegal, sorry in 2.1
};
```

Note – Release 3.0.1 correctly reports these errors.

Destructors for Built-In Types (§12.5, change)

The Release 2.1 *Reference Manual* allows explicit destructor calls for any built-in type, as in the example below. However, Release 2.1 does not implement this syntax.

```
void f(int* p) {
p->int::~int(); // legal, but error in 2.1
};
```

Note – Release 3.0.1 correctly implements this syntax.

Delete Operator (§12.6, change)

The Release 2.1 *Reference Manual* tightens the rules for the `delete` operator. Only one `operator delete()` may be declared per class, and the global `operator delete()` may not be overloaded. Release 2.1 does not enforce these restrictions.

For example, the second declaration of the `delete` operator in each scope below is illegal, but the code is accepted by both Release 2.0 and 2.1.

```
typedef unsigned int size_t;

void operator delete(void*);
void operator delete(const void*);   // error, not reported

struct S {
      void* operator new(size_t);
      void* operator new(size_t, void*);
      void operator delete(void*);
      void operator delete(void*, size_t);// error, not reported
};
```

Note – Release 3.0.1 correctly reports these errors.

Generating the Default Assignment Operator (§12.9, clarification)

The Release 2.1 *Reference Manual* states the condition under which a default assignment operator is generated differently from the old *Reference Manual.* Formerly, the condition was the following:

> "If a class `X` has any `X::operator=()` defined, even one that takes an argument of a type unrelated to `X`, `X::operator=(const X&)` will not be generated."

The Release 2.1 *Reference Manual* says

> "If a class `X` has any `X::operator=()` that takes an argument of class `X`, the default assignment will not be generated."

The new description reflects the behavior of Release 2.0 and 2.1.

Argument Matching Rules (§13.3, clarification)

Several details about the function matching rules have changed.

- In the Release 2.0 *Reference Manual* there was a rule that a call needing only standard conversions is preferred over one requiring user-defined conversions. This rule has been eliminated in the Release 2.1 *Reference Manual* and the new semantics have been implemented in Release 2.1. For example,

```
struct Complex { Complex(double); };
void f2(int, Complex);
void f2(double, double);

void y2() {
      f2(3, 4);// ambiguous
}
```

  For this code, Release 2.1 correctly reports an ambiguity.

- The second function matching change involves the treatment of arguments of type `T` that require temporaries. The Release 2.0 *Reference Manual* specified that a match with conversions requiring temporaries was a legal match. So, for example, the call to `f3(char&)` in the following code was legal and was accepted by Release 2.0:

```
void f3(char&);
void x3() {
     f3('c');
}
```

Furthermore, since standard conversions were preferred to conversions requiring temporaries, the *Reference Manual* specified that the call to `f4()` below would be resolved to `f4(int)`. Instead, Release 2.0 resolved it to `f4(char&)`:

```
void f4(int);
void f4(char&);
void x4() {
      f4('c');
}
```

Under the new rules, the calls to `f3()` and `f4()` are in error because a non-`const` reference cannot be initialized with a non-lvalue (see §8.4.3). However, Release 2.1 does not report these errors and instead preserves the Release 2.0 behavior by resolving the calls to `f3(char&)` and `f4(char&)` respectively.

Note – Release 3.0.1 correctly reports errors in this situation.

Prefix and Postfix Increment and Decrement Operators (§13.5, change)

The Release 2.0 *Reference Manual* provided no way to distinguish user-defined prefix increment and decrement operators from postfix increment and decrement operators. The Release 2.1 *Reference Manual* specifies a separate syntax for defining prefix and postfix increment and decrement operators. The prefix increment and decrement operators take one argument (the implicit `this` argument for a member function), whereas the postfix version takes two arguments (including the implicit `this` argument). For example,

```
struct S {
       operator++();  // 2.0: prefix or postfix
                      // 2.1: prefix, but not implemented as such
       operator++(int);// 2.1: postfix ++, not implemented
};
```

However, Release 2.1 does not recognize the new syntax. Use of the postfix form results in the following error message:

```
line 4: error: S:: operator ++() takes no argument
```

Note – Release 3.0.1 correctly recognizes this syntax.

ANSI C Preprocessing (§16.1, change)

The description of preprocessing in the Release 2.1 *Reference Manual* reflects the rules of ANSI C rather than of K&R C. Because the C++ Language System does *not* include a preprocessor, the actual preprocessing behavior of Release 2.1 depends on the preprocessor resident on the host machine.

New Warning Messages

"Not Used" Warning Messages Reported More Consistently

Release 2.1 issues warning messages more consistently if an object is declared but not used.

For example, Release 2.0 did not issue a "not used" message for the following code:

```
int f() {
      int array[5];
      return 0;
}
```

Release 2.1 issues a warning that `array` is not used.

Warning for Pure Virtual Destructors (§10.4, §12.5)

Release 2.1 issues a new warning if a pure virtual destructor is declared but not defined. For example, the code

```
struct B {
      // ...
      virtual ~B() = 0;
};
```

elicits the warning

```
line 1: warning: please provide an out-of-line definition: B::~B()
{}; which is needed by derived classes
```

to remind you that a definition of `B::~B()` is required.

To understand why a pure virtual destructor of an abstract class must be defined, consider what happens when a class `D` is derived from the class `B` defined above:

```
struct D : B {
      // ...
      virtual ~D();
};
D::~D() { /* ... */ }
```

The *Reference Manual* says that base class destructors are implicitly executed after the destructors for their derived classes (§12.4). This means that the compiler will generate code to call `B::~B()` at the end of `D::~D()`. Therefore `B::~B()` must have a definition; otherwise, a link-time error will occur because the definition is missing.

Why doesn't the compiler implicitly generate an empty definition for `B::~B()`? The reason is that it is legal for the user to define a `B::~B()` that is not empty! If the compiler generated an empty `B::~B()` in one compilation and the user defined a non-empty `B::~B()` in another compilation, then there would be two different definitions of the destructor. Although this inconsistency would probably be detectable at link time, it is preferable to avoid the inconsistency altogether by requiring the user to define the destructor explicitly.

Anachronism Warning Messages

Release 2.1 issues warning messages for all uses of anachronisms. Section 2.4, "Future Compatibility Issues," on page 81 describes these messages in more detail.

Library Changes

`iostream::get()` and `iostream::put()` Now Inline

The Release 2.0 version of the `iostream` library declared `iostream::get()` and `iostream::put()` to be `inline`, but both functions were too complex to be successfully inlined. The Release 2.1 implementations of these functions have been changed so that most calls can be generated inline. For example, the call to `is.get()` is inlined by Release 2.1:

```
#include <iostream.h>
void f() {
      istream is(0);
      char c;
      is.get(c);
}
```

Task Library Ported to Amdahl UTS Computers

For Release 2.1 the task library has been ported to a new platform, Amdahl UTS. To build the task library for the Amdahl UTS computer, either set `MACH=uts` in the top level `makefile` or specify it on the command line when building the task library. For example,

```
make MACH=uts libtask.a
```

`patch`

The file `BSDpatch.c` has been modified so that `patch` works under BSD Release 4.3 running on "DEC VAX" computers.

2.4 *Future Compatibility Issues*

Anachronisms

The C++ Language System provides several extensions to the C++ language to enable users to make a gradual transition from previous versions of the C++ language to the current definition, which is specified in the *Reference Manual.* In

general, these extensions allow constructs to be used that are no longer legal under the current definition, but were previously legal. The +p option disables most of these extensions so that only the "pure" language is accepted.

The following set of extensions were provided in Release 2.0 and 2.1, and most have been phased out in Release 3.0.1. Most of these extensions are listed and explained in §B.3 of the *C++ 3.0.1 Language System Product Reference Manual.* The complete list appears below, with additional references to sections of the *Reference Manual* and example programs that demonstrate each anachronism.

Release 2.1 reports uses of these extensions, except the last two, by issuing a warning message, as shown. Each of these messages has the string `(anachronism)` at the end. All of the anachronism warnings are issued unconditionally, except as noted.

In most cases, anachronisms that were warned about by default in Release 2.1 are considered errors by Release 3.0.1. Anachronisms that produced warnings only when the +w option in effect in Release 2.1 are now warnings by default and will be disallowed in the next release.

- Use of the `overload` keyword (§2.4)

```
overload f;
```

Release 2.1 – warning under the +w option

```
line 1: warning: 'overload' used (anachronism)
```

Release 3.0.1 – unconditional warning, or error if the +p option is in effect

- Use of . instead of : : for scoping (§5.1)

```
struct S {
      int f();
};
int S.f() { return 0; }
```

Release 2.1 – warning

```
line 4: warning: '.' used for qualification; please use '::'
(anachronism)
```

Release 3.0.1 – error

- Use of the `delete [n]` syntax (§5.3.4)

```
struct S { S(); ~S(); };
void f() {
      S* p = new S[10];
      // ...
      delete [10] p;
}
```

Release 2.1 – warning under the +w option

```
line 5: warning: v in 'delete[v]' is redundant; use 'delete[]'
instead (anachronism)
```

Release 3.0.1 – warning under the +p option

- Cast of a bound pointer (§5.4, §B.3.4)

```
struct S {
        int f();
} s;
typedef int (*PF)();
PF pf = (PF) &s.f;
```

Release 2.1 – warning

```
line 5: warning: address of bound function (try using ''S ::*''
                 for pointer type and ''&S ::f'' for address)
                 (anachronism)
```

Release 3.0.1 – error

- Assignment of a value of integral type to an enumeration type (§7.2)

```
enum E { e1, e2 };
void f(int i) {
        E local = i;
}
```

Release 2.1 – warning

```
line 3: warning: int assigned to enum E (anachronism)
```

Release 3.0.1 – error

- Non-`const` reference initializer not an lvalue (§8.4.3)

```
void f() {
      int& r = 5;
}
```

Release 2.1 – warning under the `+w` option

```
line 2: warning: initializer for non-const reference not an
lvalue (anachronism)
```

Release 3.0.1 – unconditional warning, or error if the `+p` option is in effect

- Non-`const` member function called for a `const` object (§9.3.1)

```
struct S {
         int f();
};
extern const S s;
int i = s.f();
```

Release 2.1 – warning

```
line 4: warning: non-const member function S::f() called for
const object (anachronism)
```

Release 3.0.1 – error

- Static data member declared within a local class (§9.4)

```
int main() {
      struct S {
            static int i;
      };
      // ...
      return 0;
}
```

Release 2.1 – warning

```
line 3: warning: static member S::i in local class S
(anachronism)
```

Release 3.0.1 – error

- Use of an unqualified nested type name outside its enclosing class definition (§9.7)

```
struct Enclosing {
      enum Nested { e1, e2 };
};

Nested var;
```

Release 2.1 – warning

```
line 5: warning: use Enclosing:: to access nested enum type
Nested (anachronism)
```

Release 3.0.1 – error

- Use of an identifier that is declared at global and local scope within a nested type definition (§9.7)

```
int i;
struct S {
      static int i;
      struct Nested {
            static int f() { return i; }
      };
};
```

Release 2.1 – warning

```
line 5: warning: i, accessed within nested class Nested, is
visible both globally and within enclosing class S -- using ::i
(anachronism)
```

Release 3.0.1 – accepted under complete nested semantics

- Use of a type name that is declared at global scope and within a local nested class (§9.7)

```
typedef int T;

void f() {
      struct Nested {
            typedef char T;
      };
      T var;
}
```

Release 2.1 – warning

```
line 7: warning: T occurs at outer and nested local class scope;
using typedef Nested::T (anachronism)
```

Release 3.0.1 – accepted under complete nested semantics

- First parameter of `operator new()` not of type `size_t` (§12.5)
- Second parameter of `operator delete()` not of type `size_t` (§12.5)

```
struct S {
        void* operator new(long);
        void operator delete(void*, long);
};
```

Release 2.1 – warning

```
line 2: warning: operator new() first argument should be size_t
(anachronism)
line 3: warning: operator delete()'s 2nd argument should be a
size_t (anachronism)
```

Release 3.0.1 – error

- `operator=()` declared as a global function (§13.4.3)

```
struct S { /* ... */ };
S& operator=(S&, S&);
```

Release 2.1 – warning

```
line 2: warning: non-member operator =() (anachronism)
```

Release 3.0.1 – error

- Use of the "Classic C" style function definition syntax (§B.3.1)

```
int f(i)
int i;
{
return i;
}
```

Release 2.1 – warning

```
line 1: warning: old style definition of f() (anachronism)
```

Release 3.0.1 – remains a warning to maintain "Classic C" compatibility

- Use of an old-style base class initializer in a constructor definition (§B.3.2)

```
class B {
      int b;
public:
      B(int i) { b = i; }
};
struct D : public B {
      D(int i) : (i) {}
};
```

Release 2.1 – warning

```
line 7: warning: name of base class B missing from base class
initializer (anachronism)
```

Release 3.0.1 – remains a warning to allow portability between Release 2.0 and Release 3.0.1.

- Assignment to `this` (§B.3.4)

```
extern void* myalloc(unsigned int);
struct X { X(); };
X::X() {
     if (this == 0) {
          this = (X*) myalloc(sizeof(X));
          // ...
     }
     else {
          this = this;
          // ...
  }
}
```

Release 2.1 – warning under the `+w` option

```
line 3: warning: assignment to this (anachronism)
```

Release 3.0.1 – unconditional warning, or error if the `+p` option is in effect

- Use of the `c_plusplus` preprocessor macro (§16.1)
- Static data member declared but never defined (§9.4)

Note – This anachronism is enforced for template classes, and will be disallowed in the next release.

The last two extensions — use of the `c_plusplus` preprocessor macro and implicit definition of a static data member — are difficult for the compiler by itself to detect, and do not produce warning messages. Uses of `c_plusplus` are generally known only to the preprocessor, and implicit definitions of static data members can only be detected at link time, or after linking has taken place. You can look for uses of `c_plusplus` by scanning your source code for that pattern. On some systems you can also find implicit definitions of static data members by examining the executable file produced by the linker for instances of uninitialized data with class-scope names.

The Old Stream Library

The old `stream` library, which is available as `lib Ostream.a` in Release 2.1, is not provided with this release of the C++ Language System.

Known Problems

The following sections describe specific problem areas that remain in the C++ Language System. Where appropriate, the related sections of the *C++ 3.0.1 Language System Product Reference Manual* are noted.

A.1 Multiple Definitions (§3.4)

- In K&R C and in the ANSI C standard, implementations are free to decide how to treat multiple, uninitialized definitions of objects with external linkage at global scope. In C++ exactly one definition, initialized or uninitialized, may occur in a single program. In order to enforce this rule, the C++ Language System initializes most global variables to `0`. However, in order to reduce object file space, no initialization is done for global arrays. Similarly, since most K&R C compilers reject such code, no initialization is done for unions or for classes or arrays of classes whose first element is a union.

  Users should be aware that invalid multiple definitions for these cases may go undetected.

- For compatibility with previous releases of the C++ Language System, static data members of non-template classes are implicitly defined. This means that multiple definitions of the same static member in multiple files will result in multiple calls to the constructor. For example, suppose that the header file `a.h` defines a class with a static member:

```
struct A { A(); };
struct B { static A ab; };
```

and file `a.c` contains the definition of the static data member:

```
#include "a.h"
A B::ab;
```

as does file `main.c`:

```
#include "a.h"
A B::ab;
main() { /* ... */ }
```

When these files are compiled and linked together, the duplicate definitions of `B::ab` will not be reported and the constructor for `B::ab` will be called twice.

A.2 Global Inline Functions Are Static (§7.2, §7.2, §3.4)

The Release 2.0 *Reference Manual* allowed a non-member inline function to have external linkage. The Release 3.0.1 *Reference Manual* specifies, however, that a name of global scope that is declared `inline` is local to its file.

Release 2.1 and Release 3.0.1 do not conform to these rules. For example, the following code is accepted by Releases 2.0, 2.1 and 3.0.1: `f()` is treated as a `static` function, and a `static` definition of `f()` is laid down.

```
extern int f(int);
inline int f(int i) { return i; }// error, not reported
int i = f(0);
int (*pf)(int) = &f;
```

Instead, the C++ Language System should report an error that f() cannot be redeclared as inline after being declared extern.

A.3 Reuse of a Class Name by its Members (§9.3)

The Release 3.0.1 *Reference Manual* limits the ways in which a class name can be reused by members of the class. The rule is that a static data member, enumerator, member of an anonymous union, or nested type may not have the same name as its class.

Release 3.0.1 does not enforce these restrictions completely. An error is reported if an enumerator or nested type has the same name as its enclosing class, but a static data member or member of an anonymous union are not caught.

```
struct S1 {
static int S1;// illegal, no error
};
struct S2 {
union { int i; float S2; };// illegal, no error
};
```

A.4 Unions (§9.4)

The C++ Language System invalidly allows union members of a type which contains a user defined assignment operator. It correctly detects union members of a type with a constructor or destructor:

```
struct assign {
//...
assign& operator =(const assign&);
};
struct ctor {
//...
ctor();
};
struct dtor {
//...
~dtor();
};
union U {
assign a;// undetected error
ctor b;// correctly detected
dtor c;// correctly detected
};
```

The following correct errors are reported

```
line 16: error: member U::b of class ctor with constructor in
        union
line 16: error: member U::c of class dtor with destructor in union
```

but there should be a similar error

```
line 16: error: member U::a of class assign with operator= in union
```

A.5 Nested Types (§9.6)

This release completes the introduction of true nested types. There are two known problems in the new implementation:

- The C++ Language System generates invalid C code for uses of nested classes as virtual base classes:

```
struct Outer {
struct InnerBase {
//...
};
struct InnerDerived : public virtual Outer::InnerBase {
//...
};
};
```

- Protection has not yet been implemented for nested types:

```
class A {
enum E {/*...*/}; // private
//...
};
A::E evar;// undetected error,
// A::E should not be accessible
```

A.6 *Pure Virtual Functions (§10.4)*

- The C++ Language System fails to detect the use of a pure virtual function inside the class's own destructor. Other invalid uses of a pure virtual function are correctly detected:

```
struct Base {
Base();
~Base();
virtual void f() =0;
};
Base::~Base() {
f();        // undetected error
};
Base::Base() {
f();        // correctly detected
};
Base f();   // correctly detected
f(Base);    // correctly detected
```

The following errors are correctly reported

```
line 13: error: call of pure virtual function Base::f() in
constructor Base::Base()
line 15: error: abstract class Base cannot be used as a function
return type
line 15: Base::f() is a pure virtual function of class Base
line 16: error: abstract class Base cannot be used as an argument
type
line 16: Base::f() is a pure virtual function of class Base
```

but there should be a similar error reported on for the case involving the destructor.

A.7 Friendship (§11.5)

The C++ Language System invalidly extends friendship throughout the class hierarchy in a multiple inheritance lattice:

```
class base1 {
friend void foo();
protected:
int i;
};
class base2 {
protected:
int j;
};
class derived : public base1, public base2 {
protected:
int k;
public:
derived();
};
void foo() {
derived der;
der.i = 1;// ok, foo is friend of base1
der.j = 2;// undetected error
der.k = 3;// detected error
};
```

The following correct error is reported:

```
Line 23: error: foo() cannot access derived::k: protected member
```

but there should be a similar error for the assignment to `der.j`:

```
Line 22: error: foo() cannot access derived::j: protected member
```

A.8 Static Members (§11.6)

- Release 3.0.1 is too restrictive in its treatment of protected static members of a base class when they are accessed by friends of a derived class. The following example should compile without complaint:

```
class S1 {
protected:
static int s;
};
struct S2 : public S1 {
friend int f1() { return S1::s; }// legal
friend int f2() { return S2::s; }// legal
};
```

Instead, the following errors are incorrectly reported:

```
error: f2() cannot access S1::s: protected member
error: f1() cannot access S1::s: protected member
```

This problem can be circumvented by referring to the base class's static member through an object of the derived class:

```
friend int f3(const S2& s2) { return s2.s; }
```

- Section 11.5 of the *C++ 3.0.1 Language System Product Reference Manual* states that "a friend or a member function of a derived class can access a protected static member of a base class" and section 12.5 specifies that "An X::operator new() [delete()] for a class X is a static member." The C++ Language System fails to allow the implied access to static members new and delete:

```
typedef unsigned int size_t;
class base {
protected:
void * operator new(size_t);
void operator delete(void *);
void static_memf();
};
class derived : public base {
public:
void f() {
base *b = new base();// invalidly rejected
delete b;// invalidly rejected
static_memf();// correctly allowed
};
};
```

Produces the following invalid errors:

```
line 10: error: derived::f() cannot access base::operator
delete(): protected member
line 10: error: derived::f() cannot access base::operator new():
protected member
```

A.9 Access control and constructors and destructors (§12.4)

- The reference manual stipulates that normal access control is applied to constructors and destructors. This implies that making a destructor private or protected disallows automatic and static allocation of such objects since they could never be destroyed. The C++ Language System correctly enforces this rule in most situations. However, it invalidly creates temporaries of such types when passing arguments as const references and then invalidly calls the private destructor:

```
class A;
struct B {
B();
~B();
void foo (A const&);
};
class A {
private:
~A();
void operator=(A&);
A(A&);
public:
A(B);
};
main() {
B b;
A a(b);         // correctly detected
A a1 = A(b);    // correctly detected
b.foo(b);       // undetected error
b.foo(A(b));    // undetected error
};
```

The following correct errors are reported:

```
line 21: error: main() cannot access A::~A(): private member
line 22: error: main() cannot access A::~A(): private member
```

but there should be similar errors for the calls to b.foo.

- The C++ Language System also fails to detect invalid calls to operator delete for classes with a private destructor:

```
class B {
~B();// private
};
main() {
B b;// correctly detected
B* bp = new B;// legal
delete bp;// undetected error
};
```

A.10 Protection and Destructors (§12.5)

- If a base class has a private destructor, only member and friend functions of that class may destroy objects of that class. However, Release 3.0.1 fails to enforce this protection for derived classes that do not redefine the destructor at the same protection level. Thus, protection can be overridden by a derived class that simply fails to declare a destructor or by a derived class that declares a destructor with less restrictive protection. For example, the following code compiles without complaint:

```
class B {
private:
~B();
};
class D: public B {};
class D2: public B {
public:
~D2() { }
};
void f() {
D d;    // undetected error
D2 d2; // undetected error
}
```

Instead, `f()` should not be able to create objects of type `D` or `D2`.

A.11 Template Formal Type Declarations (§14.2)

- The C++ Language System is unable to parse some complicated formal template arguments. For example, function pointer arguments are invalidly rejected. Typedefs can be used to work around such problems:

```
template <void (*fp)()> struct A {};// invalidly rejected
typedef void (*T)();
template <T p> struct B {// correctly accepted
T var;
};
```

- It is possible to construct suitably complicated function names that internal name limits are overrun. This can happen when generating function signatures with template type arguments which use literal formal arguments:

```
struct Application_area {};
template <class T, int i> struct Vector {
T data[i];
};
typedef Vector<Application_area,37> T1;
typedef Vector<Application_area*,47> T2;
typedef Vector<Application_area&,89> T3;
template <class T, class U, class V> struct A {};// ok
A<T1,T2,T3> a;// ok
void f(A<T1,T2,T3>) {}// bad c generated
```

A.12 *Template Classes (§14.3)*

- In processing templates, the C++ Language System builds up internal representations of template classes and functions but does not type check or otherwise validate user code until a template is instantiated. For example, in the code below, the definition of the non-existent static member `y` is not detected until a variable of the template type `A` is declared:

```
template <class T> struct A {
static int x;
};
template <class T> int A<T>::y = 37; // error not detected until
// a variable of type A<...>
// is declared
```

It is a good idea, therefore, when developing code that defines template classes or functions to include simple references to the template type to force instantiation time type checking and other semantic checking. For example, if the above code had been compiled with a use of template A, the error would have been correctly reported:

```
template <class T> struct A {
static int x;
};
template <class T> int A<T>::y = 37;
A<int> _dummy;
```

Produces the following correct error messages:

```
line 8: error: y: only static data members can be parameterized
```

A.13 Template Declarations(§14.6)

If two class templates refer to each other, one referring to the other only via a pointer or reference, and the other referring to the first in a way that requires the full definition to be known, the C++ compiler may produce errors depending on the order in which uses of the templates appear in user code. For example:

```
template <class T> class B;
template <class T> class A {
B<T>* ptr;
};
template <class T> class B {
A<T> not_a_ptr;
};
A<int> something; //Causes error
```

produces the following invalid errors:

```
line 9: error: A undefined, size not known
line 9: error detected during the instantiation of B <int >
line 9: the instantiation path was:
line 3: template: B <int >
line 12: template: A <int >
```

If a use of `A<int>` is seen before a use of `B<int>`, the instantiation will either fail, or produce invalid C code. If a reference to `B<int>` is seen first, there are no errors. The workaround is to add a dummy reference to `B<int>` before the first reference to `A<int>`:

```
typedef B<int> dummy;
A<int> something;
```

A.14 Member Function Templates (§14.7)

- In processing templates, the C++ Language System builds up an internal representation for the template, but does not actually process instantiations until the end of the file. This is to allow for correct processing of template specializations. However, this approach has several side effects with respect to processing inline member function templates. To be inlined, member functions must be defined inside the class definition. For example, the Vector constructor in the following code will be laid down out of line in each file rather than being inlined:

```
template<class T> class Vector {
public:
inline Vector(int size);
T& operator[](int i) {return vec[i];}
private:
int size;
T* vec;
};
template<class T>
inline Vector<T>::Vector(int sz) : vec(new T[size=sz]) {}
main()
{
typedef char *String;
```

```
template<class T> class Vector {
String a = "foo_bar";
Vector<String> str_vec(2);
str_vec[0] = a;
}
```

Similarly, errors will be reported if a member function is not declared to be inline in the class template but is but subsequently defined as inline. For example:

```
template <class T> class A {
public:
void f(T t);
};
template <class T> inline void A<T>::f(T t)
{
}
main()
{
A<int> a;
a.f(0);
}
```

produces the following errors:

```
line 7: error: A <int>::f() declared with external linkage and
called before defined as inline
line 7: error detected during the instantiation of A <int >
line 24: is the site of the instantiation
```

- Specializations of template functions that are friends are not recognized as friends if the friend declaration is not itself specialized:

```
template <class T> class A {
friend void f(A<T>&); // each f is a friend of A<T>
int i;
};
template <class U>    // template instance of f
void f( A<U> &a )
{
int j = a.i; // ok
}
```

```
template <class T> class A {
void f(A<int> &a)      // define a specialization for f
{
int j = a.i;           // spurious error: A<int>::i private
}
```

Produces the following spurious error:

```
line 14: error: f() cannot access A <int >::i: private member
```

The workaround is to include a friend declaration for each specialization:

```
template <class T> class A2 {
friend void f2(A2<T>&);// each f is a friend of A<T>
// specializations must be explicitly declared friends
friend void f2(A2<int>&);
int i;
};
template <class U>
void f2( A2<U> &a )
{
int j = a.i;
}
void f2(A2<int> &a)
{
int j = a.i;// ok
}
```

A.15 *Incompatibilities with the ANSI C Standard*

- Release 3.0.1 fails to accept ANSI C-conforming declarations for functions taking function arguments. For example,

  ```
  void f(int());
  ```

  produces the following error:

  ```
  line 1: error: bad base type: void f
  ```

  If a function pointer is specified as a parameter, like this,

  ```
  void f(int(*)());
  ```

  the code is accepted.

A.16 Missing or Extraneous Warnings

- The C++ Language System is sometimes too cautious in deciding when it is necessary to generate code to invoke a destructor. As a result, unreachable code containing destructor invocations is sometimes generated, and some C compilers warn about this unreachable code. For example:

```
struct A {
A();
~A();
};
void f(int i) {
switch(i) {
case 0: {
A a;
break;
};
}
}
```

  This code may result in the C compiler warning

  ```
  line 6: warning: statement not reached
  ```

  which can be safely ignored.

- Similarly, for the following case, destructors are properly called on each return path, but also at the end of the function:

```
struct A {
A();
~A();
};
int f(int i) {
A a;
if (i)
return i;
else
return i;
}
```

- C++ allows conversions that may involve loss of information. Because such conversions are likely to introduce errors in the user's code, the C++ Language System should warn about shortening conversions. In general, such conversions are diagnosed only when assigning a `float`, `double`, or `long` value to one of the smaller integral types. The following shortening conversions are accepted without complaint:

```
extern char c;
extern short s;
extern unsigned char uc;
extern unsigned short us;
extern int i;
extern long l;
extern float f;
extern double d;
void x() {
f = d;
c = i;
l = d;
l = f;
c = s;
s = i;
uc = i;
us = i;
}
```

- Some instances of "used before set" warnings are invalid. For example, the code below causes the C++ Language System to warn incorrectly that `s` is used before set.

```
struct S {
short a;
short b;
};
void f() {
S s;
s.a = s.b = 0;// invalid ''used before set'' warning
}
```

Use of the `sizeof` operator also leads to invalid "used but not set" warnings, as in the following code:

```
void g() {
char *p;
int i = sizeof(p);//invalid ``used but not set'' warning
}
```

A.17 Other Problems with Compiling and Linking

- Single files compiled directly to an `a.out` which contain specializations of templates will occasionally fail. For example:

```
extern "C" void printf(...);
template <class T> int foo(T) { return 1; }
main()
{
int i = 3;
printf("%d\n", foo(i)); // should print 0
}
int foo(int) { return 0; }
```

Invalidly produces the following error from the C compiler:

```
line 11: redeclaration of foo__Fi
```

This is because the single file case is optimized to avoid automated instantiation support and the system invalidly instantiates the `printf` call to create the template version of `foo(int)`. When the subsequent

specialization is seen, the template instance has already been created. This problem can be avoided either by ensuring that all specializations precede any use:

```
extern "C" void printf(...);
template <class T> int foo(T) { return 1; }
int foo(int) { return 0; }
main()
{
int i = 3;
printf("%d\n", foo(i)); // should print 0
}
```

or by compiling `CC -ptn` which ensures that full automated instantiation support is invoked.

- At present, function templates declared in header files have only the raw name extracted and added to the name mapping files. So for:

  ```
  template <class T, class U> void f(T, int, U);
  ```

  `f` will be extracted. This works for many simple cases, but fails in some cases where two different headers declare a function template with the same or different formal arguments.

- For compatibility with previous releases, the argument type of `__vec_new` and `vec_delete` under the +a1 ANSI option are incorrect. Strict ANSI compliance would declare these functions as:

```
__vec_new(void*, int, int, void(*)());
__vec_delete(void *, int, int, void(*)(), int, int);
```

However, doing so would lead to bootstrapping problems when building the compiler using `libC` compiled with earlier releases.

- For compatibility with previous releases, static class members and static template class members created from a template specialization are not initialized to `0`. This may lead to problems with linkers that do not pull in object files from an archive if there are no initialized external references. If a file exists whose only external dependency is an uninitialized static data

member or an uninitialized global array, these linkers will fail to include the object file and a runtime error will occur. For example, suppose `ab.h` defines a class with a static data member:

```
// file "ab.h"
struct A {
int i;
A() { i = 3; }
};
struct B {
static A a;
};
```

and file `ab.c` defines the static member

```
// file "ab.c"
#include "ab.h"
A B::a;
```

and file `main.c` refers to the static member

```
// file "main.c"
#include <stdio.h>
#include "ab.h"
main() {
printf ("%d\n", B::a.i);
// ...
return 0;
}
```

If these files are directly compiled and linked, the expected output of `3` is printed on the standard output. However, if the file `ab.c` is compiled and stored in a library and later linked with `main.c`, then the program prints `0` if a linker that does not resolve uninitialized data is used.

- When two files are compiled separately in separate directories, but contain identically named objects of the same class, problems will occur when an attempt is made to link the two object files. For example, suppose you have a header file `x.h` in your current directory, and you have two subdirectories `a` and `b`, each of which contains a file named `x.c`.

```
// file "x.h"
struct X {
virtual void f() {};
};
// file "a/x.c"
#include "../x.h"
void f() {
X x;
}
// file "b/x.c"
#include "../x.h"
main() {
X x;
// ...
return 0;
}
```

If these files are compiled separately and an attempt is made to link them together,

```
cd a
CC -c x.c
cd ../b
CC -c x.c
cd ..
CC a/x.o b/x.o
```

they will fail to link, and messages similar to the following will be generated by the linker:

```
ld: Symbol _
'.01c'_
'.01c'_vtbl_
'.01c'_1X_
'.01c'_x_c in b/x.o is multiply defined. First defined in a/x.o
ld: Symbol _
'.01c'_
'.01c'_ptbl_
'.01c'_1X_
'.01c'_x_c in b/x.o is multiply defined. First defined in a/x.o
ld fatal: Error(s). No output written to a.out
```

These errors occur because the names of the virtual tables and associated housekeeping information for the `X` objects in files `a/x.c` and `b/x.c` are encoded identically, so the symbols are multiply defined.

A workaround for this problem is to rename one of the files or to use a longer pathname when compiling these files.

A.18 Library Problems

The implementation of the task library limits the number of levels of derivation from class `task` to one. That is, a class derived from class `task` may not have derived classes. However, use of multi-level inheritance is not detected and usually results in an unexpected runtime core dump. One possible workaround

for this limitation is to put the required complex structures in a class not derived from `task`. Then derive a trivial class from `task` whose constructor executes the coroutine in the complex task. For example:

```
class Task_base {
virtual int Main();
};
class Runner : public task {
Task_base*p;
public:
Runner(Task_base*);
};
Runner::Runner(Task_base* fp) : p(fp)
{
resultis(p->Main());
}
```

Class `Task_base` is the base class from which the user should derive whatever additional classes and structures are needed.

Implementation Specific Behavior

This appendix describes implementation specific behavior of the Sun C++ Language System. Implementation specific behaviors can be categorized as follows:

1. Behavior that the *Reference Manual* defines as "implementation dependent"
2. Behavior that depends on the ANSI-C compiler `acomp` (supplied with Sun C++) or preprocessor used with Release 3.0.1
3. Properties that are defined in the standard header files `stddef.h`, `limits.h`, and `stdlib.h`
4. Translation limits
5. Language constructs that are not implemented in this release

This appendix addresses categories 1, 2, 4, and 5. For details about properties defined in the standard header files (category 3), see the headers themselves. Additional information about constructs that are not implemented is provided in Appendix C, "Not Implemented Messages," which contains an alphabetical listing of the "not implemented" error messages.

The ordering and numbering of sections in this appendix corresponds to the order and numbering of the related sections in the *Reference Manual.* The section entitled "Translation Limits" (which does not have a corresponding section in the *Reference Manual*) precedes the numbered sections.

B.1 Translation Limits

Release 3.0.1 of the Sun C++ Language System imposes the following translation limits:

- 50 nesting levels of compound statements
- 10 nesting levels of linkage declarations
- 4088 characters in a token
- 22222 virtual functions in a class
- 10000 identifiers generated by the implementation

These limits can be changed by recompiling the translator. Additional translation limits may be inherited from the ANSI-C compiler acomp (supplied with Sun C++) and preprocessor.

B.2 Identifiers (§2.4)

Release 3.0.1reserves identifiers that contain a sequence of two underscores for its own use. In addition, identifiers reserved in the ANSI C standard are also reserved by Release 3.0.1. Under the `+w` option, identifiers with double underscores result in a warning in Release 3.0.1

B.3 Character Constants (§2.6)

The *Reference Manual* states that the value of a multi-character constant, such as `'abcd'`, is implementation dependent. Release 3.0.1 passes these constants to the ANSI-C compiler `acomp` (supplied with Sun C++), which determines their values. A multi-character constant containing more characters than `sizeof(int)` is reported as an error by Release 3.0.1.

The *Reference Manual* states that the value of a character constant is implementation dependent if it exceeds that of the largest `char`. Release 3.0.1 accepts octal and hexadecimal character literals that do not fit in a `char`. It uses the low order bits that make up the value of the constant. For example, the octal character constant `'\777'`, is treated as `'\377'`. The hexadecimal character constant `'\x123'` is treated as `'\x23'`.

Release 3.0.1 does not implement wide character constants, such as `L'ab'`. A "not implemented" error message is reported.

B.4 Floating Constants (§2.6)

When compiling with the `+a0` option, Release 3.0.1 removes an `l` or `L` suffix from a floating constant before passing the constant to the ANSI-C compiler `acomp`. Under the `+a1` option such a constant is passed unchanged to the ANSI-C compiler `acomp` (supplied with Sun C++). In either case, the constant is considered to be of type `long double` for purposes of resolving overloaded function calls.

B.5 String Literals (§2.6)

The *Reference Manual* states that it is implementation dependent whether all string literals are distinct. Release 3.0.1 does not attempt to detect cases where string literals could be represented as overlapping objects. The ANSI-C compiler `acomp` may, however, detect such cases and attempt to overlap their storage.

Release 3.0.1 does not implement wide character strings, such as `L"abcd"`. A "not implemented" error message is reported.

B.6 Start and Termination (§3.5)

The *Reference Manual* states that the type of `main()` is implementation dependent. Release 3.0.1 itself does not impose any restrictions on the type of `main()`, but the ANSI-C compiler `acomp`, or the target environment may impose such restrictions.

The Sun C++ Language System treats `main()` as if its linkage were `extern "C"`.

B.7 Fundamental Types (§3.7)

Release 3.0.1does not implement the type specifier `signed`; it issues a warning and proceeds as though the specifier `signed` had not appeared.

When Release 3.0.1 is invoked with the `+a0` option, the type `long double` is considered to be the same size and precision as the type `double` in the ANSI-C compiler `acomp`. Under the `+a1` option, `long double` is passed to the

ANSI-C compiler `acomp` as `long double`. In either case, type `long double` is considered a distinct type for purposes of resolving overloaded function declarations and invocations.

Release 3.0.1 does not impose any alignment restrictions when allocating objects of a particular type. Such restrictions, if they exist, are enforced by the ANSI-C compiler `acomp`.

B.8 Integral Conversions (§4.3)

When a value of an integral type is converted to a signed integral type with fewer bits in the representation, Release 3.0.1 issues a warning message if the `+w` option is specified. The runtime behavior of such a conversion depends on the treatment of the conversion by the ANSI-C compiler `acomp`.

B.9 Expressions (§5.1)

The *Reference Manual* states that the handling of overflow and divide check in expression evaluation is implementation dependent. When the second operand of a division or modulus operator is known to be zero at compile time, Release 3.0.1 reports an error. Overflow and other divide check conditions are handled by the ANSI-C compiler `acomp` and execution environment.

B.10 Function Call (§5.3)

The *Reference Manual* states that the order of evaluation of arguments to a function call is implementation dependent; similarly, the order of evaluation of the postfix expression, which designates the function to be called, and the argument expression list are implementation dependent. In both cases the order depends on the treatment by the ANSI-C compiler `acomp`.

B.11 Explicit Type Conversion (§5.5)

The *Reference Manual* states that the value obtained by explicitly converting a pointer to an integral type large enough to hold it is implementation dependent. This behavior is defined by the ANSI-C compiler `acomp`. Similarly, the behavior when explicitly converting an integer to a pointer depends on `acomp`.

B.12 Multiplicative Operators (§5.7)

The *Reference Manual* states that the sign of the result of the modulus operator is non-negative if both operands are non-negative; otherwise, the sign of the result is implementation dependent. This behavior depends on the ANSI-C compiler `acomp` except when the values of both operands are known at compile time. In this case, the sign of the result is the same as the sign of the numerator.

B.13 Shift Operators (§5.9)

The *Reference Manual* states that the result of a right shift when the left operand is a signed type with a negative value is implementation dependent. This behavior depends on the ANSI-C compiler `acomp`.

B.14 Relational Operators (§5.10)

According to the *Reference Manual,* certain pointer comparisons are implementation dependent. For Release 3.0.1, the results of these comparisons depend on the ANSI-C compiler `acomp`.

B.15 Storage Class Specifiers (§7.2)

The *Reference Manual* states that the `inline` specifier is a hint to the compiler. Chapter 8 of the *C++ 3.0 Language System Selected Readings* describes the treatment of `inline` functions.

When compiling with the `+d` option, Release 3.0.1 always generates out-of-line calls to inline functions.

B.16 Type Specifiers (§7.2)

Release 3.0.1 does not implement volatile type specifiers for functions; a "not implemented" error message is issued.

B.17 Asm Declarations (§7.4)

Release 3.0.1 passes `asm` declarations to the ANSI-C compiler `acomp` without modification.

B.18 Linkage Specifications (§7.5)

Release 3.0.1supports linkage to C and C++.

The effect of a `"C"` linkage specification (`extern "C"`) on a function that is not a member function is that the function name is not encoded with type information, as is otherwise done for C++ functions. Member functions are not affected by linkage specifications.

The C linkage specification (`extern "C"`), when applied to a non-function declaration, does not affect the C code generated.

B.19 Class Members (§9.3)

The *Reference Manual* states that the order of allocation of non-static data members across *access-specifiers* is implementation dependent. Release 3.0.1 allocates non-static data members in declaration order.

B.20 Bit-Fields (§9.7)

The *Reference Manual* states that the allocation and alignment of bit-fields within a class object is implementation dependent. Responsibility for the allocation and alignment of bit-fields rests with the ANSI-C compiler `acomp`.

Whether the high-order bit position of a "plain" `int` bit-field is treated as a sign bit depends on the behavior of `acomp`.

B.21 Multiple Base Classes (§10.2)

The *Reference Manual* states that the order in which storage is allocated for base classes is implementation dependent. For non-virtual base classes, Release 3.0.1 allocates storage in the order that they are mentioned in the derived class declaration.

B.22 Argument Matching (§13.3)

The type of the result of an integral promotion (§4.1) depends on the execution environment, as does the type of an unsuffixed integer constant (§2.5.1). Consequently, the determination of which overloaded function to call may also depend on the execution environment, as illustrated by an example in §13.2 of the *Reference Manual.*

B.23 Exception Handling (experimental) (§15.1)

Release 3.0.1does not implement exception handling. The keyword `catch` is reserved for future use. A "not implemented" error message is reported if `catch` is seen.

B.24 Predefined Names (§16.11)

The following macros are defined by Release 3.0.1:

`_cplusplus` The decimal constant 1.

`c_plusplus` The decimal constant 1. This macro is provided for compatibility with previous releases and will *not* be supported in the next major release. Other macros may be predefined by the underlying preprocessor.

B.25 Anachronisms (§B.4)

For compatibility with previous releases, Release 3.0.1 supports the anachronisms described in this appendix. These anachronisms will *not*, however, be supported in the next major release of the Sun C++ Language System. Chapter 2, "Compatibility," describes current and future behavior.

Not Implemented Messages

This appendix contains the text and explanation for all "not implemented" messages produced by the Sun C++ Language System Release 3.0.1. They are listed here in alphabetical order.

Each message is preceded by a file name and line number. The line number is usually the line on which a problem has been diagnosed.

A "not implemented" message is issued when Release 3.0.1 encounters a *legal* construct for which it cannot generate code. Because code is not generated, "not implemented" messages cause the `CC` command to fail, and the program is not linked. Release 3.0.1 does, however, attempt to examine the rest of your program for other errors.

- `actual parameter expression of type string literal`

  A template is instantiated with a sting literal actual argument:

```
template <char* s> struct S {/*...*/};

S<"hello world"> svar;
```

```
"file", line 3: not implemented: actual parameter expression of
type string literal
```

- `address of bound member as actual template argument`

  A template is instantiated with the address of a class member bound to an actual class object:

```
template <int *pi> class x {};
class y { public: int i; } b;

x< &b.i > xi;
```

```
"file", line 4: not implemented: address of bound member (& ::b .
y::i) as actual template argument
```

- `& of` *op*

  This message should not be produced.

- `1st operand of .* too complicated`

  The first operand of a function call expression involves a pointer to a member function and is an expression that may have side effects or may require a temporary.

```
struct S { virtual int f(); };
int (S::*pmf)() = &S::f;
S *f();
int i = (f()->*pmf)();
```

```
"file", line 5: not implemented: 1st operand of .* too complicated
```

- `2nd operand of .* too complicated`

  The second operand of a pointer to member operator is an expression that has side effects.

```
struct S { int f(); };
int (S::*pmf)() = &S::f;
S *sp = new S;
int i = 5;
int j = (sp->*(i+=5, pmf))();
```

```
"file", line 5: not implemented: 2nd operand of .* too complicated
```

- `call of virtual function before class has been completely declared.`

```
class x {
public:
virtual x& f();
int foo(x t = pt->f());
private:
static x* pt;
int i;
};
```

```
"file", line 6: not implemented: call of virtual function x::f()
before class x has been completely declared - try moving call from
argument list into function body or make function non-virtual
```

- `cannot expand inline` *function* `with for statement in inline`

  A `for` statement appears in the definition of an inline function.

```
struct S {
        int s[100];
        S() { for (int i = 0; i < 100; i++) s[i] = i; }
};
```

```
"file", line 1: not implemented: cannot expand inline function
S::S() with for statement in inline
```

- cannot expand inline function *function* with statement after "return"

  A value-returning inline function contains a statement following a return statement.

```
inline int f(int i) {
        if (i) return i;
        return 0;
}
```

```
"file", line 4: not implemented: cannot expand inline function f()
with statement after "return"
```

- cannot expand inline function *function* with two local variables with the same name *(name)*

  Two variables with the same name and different types are declared within the body of a value-returning inline function.

```
inline int f(int i) {
        { int x = i; }
        { double x = i; }
        return 0;
}
```

```
"file", line 5: not implemented: cannot expand inline function f()
with two local variables with the same name (x)
```

- `cannot expand inline function needing temporary variable of array type`

  An inline function that contains a local declaration of an array object is called.

```
inline int f(int i) {
        int a[1];
        a[0] = i;
        return i;
}
int v = f(0);
```

```
"file", line 6: not implemented: cannot expand inline function
needing temporary variable of array type
```

- `cannot expand inline function with return in if statement`

  This message should not be produced.

- `cannot expand inline function with static` *name*

  An inline function contains the declaration of a static object.

```
inline void f() {
static int i = 5;
}
```

```
"file", line 2: not implemented: cannot expand inline function with
static i
```

- `cast of non-integer constant`

  A cast of a non-integer constant as an actual parameter to a template class.

```
template <int i> class x;
int yy;

x< (int)&yy > xi;
```

```
"file", line 4: not implemented: cast of non-integer constant
```

- cannot expand inline void *function* called in comma expression

  A call of an inline void function that cannot be translated into an expression (that is, one that includes a loop, a goto, or a switch statement) appears as the first operand of a comma operator.

```
int i;
inline void f() { for (;;) ; }
void g() { for (f(), i = 0; i < 10; i++) ; }
```

```
"file", line 3: not implemented: cannot expand inline void f()
called in comma expression
```

- cannot expand inline void *function* called in for expression.

  A call of an inline void function that cannot be translated into an expression (that is, one that includes a loop, a goto, or a switch statement) appears in the second expression of a for statement.

```
void inline f() { for (;;) ; }
void g() { for (;; f()) ; }
```

```
"file", line 2: not implemented cannot expand void f() called in
for expression
```

- cannot expand value-returning incline *function* with call of ...

  A value-returning inline function is defined, and it contains a call to another inline function that is not value-returning.

```
inline void f() { for(;;) ; }
inline int g() { f(); return 0; }
```

```
"file", line 2: not implemented: cannot expand value-returning
inline g() with call of non-value-returning inline f()
```

- `cannot merge lists of conversion functions`

  A derived class with multiple bases is declared and there are conversion operators declared in more than one of the base classes.

```
struct B1 {
        operator int();
};
struct B2 {
        operator float();
};
struct D : public B1, public B2 { };
```

```
"file", line 7: not implemented: cannot merge lists of conversion
functions
```

- `catch`

  The keyword `catch` appears; `catch` is reserved for future use.

```
int catch;
```

```
"file", line 1: not implemented: catch
"file", line 1: warning: name expected in declaration list
```

- `class defined within sizeof`

  A class or union definition appears as the type name in a `sizeof` expression.

```
int i = sizeof (struct S { int i; });
```

```
"file", line 1: not implemented: class defined within sizeof
"file", line 1: error: S undefined, size not known
```

- `class hierarchy too complicated`

  This message should not be produced.

- `conditional expression with` *type*

  The second and third operands of a conditional expression are member functions or pointers to members.

```
struct S { int i, j; };
void f(int i) {
      int S::*pmi = i ? &S::i : &S::j;
}
```

```
"file", line 3: not implemented: conditional expression with int
S::*
```

- `constructor needed for argument initializer`

  The default value for an argument is a constructor or is an expression that invokes a constructor.

```
struct S { S(int); };
int f(S = S(1));
int g(S = 5);
```

```
"file", line 2: not implemented: constructor as default argument
"file", line 3: not implemented: constructor needed for argument
initializer
```

- `copy of` *member* `[], no memberwise copy for` *class*

  An implementation-generated copy operation for a class `X` is required, but the operation cannot be generated because `X` has an array member whose type is a class with either a virtual base class or its own defined copy operation. The workaround is to add a memberwise copy operator to `X`.

```
struct S1 {};
struct S2 : S1 { S2& operator=(const S2&); };
struct X { S2 m[1]; };
X var1;
X var2 = var1;
```

```
"file", line 5: not implemented: copy of S2[], no memberwise copy
for S2
```

- `default argument too complicated`

  A default argument in a declaration not at file scope requires the generation of a temporary.

```
struct S {
      S();
      int f(const int &r = 1);
};
```

```
"file", line 3: not implemented: default argument too complicated
"file", line 3: not implemented: needs temporary variable to
evaluate argument initializer
```

- `ellipsis(...) in argument list of template function` *name*

  An ellipsis is used in a template function declaration:

```
template <class T> f(T, ...);
```

```
"file", line 1: not implemented: ellipsis (...) in argument list of
template function f()
```

- `explicit template parameter list for destructor of specialized template class` *name*

  Explicit template parameters are included in declaration of a specialized class' destructor:

```
template <class T> struct S { /*...*/ };

struct S<int> {
      ~S<int>();
};
```

```
"file", line 4: not implemented: explicit template parameter list
for destructor of specialized template class S <> -- please drop
the parameter list
```

Instead, declare the specialized destructor as follows:

```
template <class T> struct S { /*...*/ };

struct S<int> {
      ~S();
};
```

- `formal type parameter` *name* `used as base class of template`

  The formal type parameter is used as the base class of a template class:

```
template <class T> struct S : public T {/*...*/};
```

```
"file", line 1: not implemented: formal type parameter T used as
base class of template
```

- `forward declaration of a specialized version of template` *name*

  A forward declaration of a specialized, rather than generalized template:

```
template <class T> struct S;
struct S<int>;
```

```
"file", line 2: not implemented: forward declaration of a
specialized version of template S <int >
```

- `general initializer in initializer list`

  The initializer list in a declaration contains an expression that cannot easily be evaluated at compile time or that requires runtime evaluation.

```
int f();
int i[1] = { f() };
```

```
"file", line 2: not implemented: general initializer in initializer
list
```

- `initialization of` *name* `(automatic aggregate)`

  An aggregate at local scope is initialized. This message is not issued if the `+a1` option (produces declarations acceptable to an ANSI C compiler) is specified.

```
void f() {
int i[1] = {1};
}
```

```
"file", line 2: not implemented: initialization of i (automatic
aggregate)
```

- `initialization of union with initializer list`

  An object of union type is initialized with an initializer list. This message is not issued if the `+a1` option (produces declarations acceptable to an ANSI C compiler) is specified.

```
union U { int i; float f; };
U u = {1};
```

```
"file", line 2: not implemented: initialization of union with
initializer list
```

- `initializer for class member array with constructor`

  This message should always be accompanied by an error message. The "not implemented" message is inappropriate and should not be reported.

- `initializer for local static too complicated`

  This message should not be produced.

- `initializer for multi-dimensional array of a objects of a class` *class* `with a constructor` *name*.

  A multi-dimensional array of a class with a constructor has an explicit initializer.

```
struct S { S(int); };
S s[2][2] = {1,2,3,4};
```

```
"file", line 2: not implemented: initializer for multi-dimensional
array of objects of class S with constructor ::s
```

- `implicit static initializer for multi-dimensional array of objects of class with constructor`

```
class x {
public:
        x() ;
};

main() {
static x xx[10][20];
}
```

```
"file", line 7: not implemented: implicit static initializer for
multi-dimensional array of objects of class x with constructor
```

- `initializer list for local variable` *name*

  This message should not be produced.

- `label in block with destructors`

  A labeled statement appears in a block in which an object with a destructor exists.

```
struct S { S(int); (apS(); };
void f() {
      S s(5);
xyz:;
}
```

```
"file", line 5: not implemented: label in block with destructors
```

- `local class` *name* `within template function`

  A local class is defined inside a template function. A similar message is issued for `local` enums and `local` typedefs defined inside a template function:

```
template <class T> f() {
class l {/*...*/};
enum E {/*...*/};
typedef int* ip;
};
```

```
"file", line 2: not implemented: local class l (local to f()) within
template function
"file", line 3: not implemented: local enum E(local to f()) within
template function
"file", line 4: not implemented: local typedef ip within template
function
```

- `local static class` *name* (*type*)

  A static array of objects of a class with a constructor is declared at local scope.

```
class S {
public:
    S();
};
void f() {
    static S s[9];
}
```

```
"file", line 2: not implemented: local static class s ( S [9])
```

- `local static name has` *class*::~*class*() `but no constructor (add` *class*::*class*())

  A static class object with a destructor, but no constructor, appears at local scope.

```
struct S { ~S(); };
void f() { static S s; }
```

```
"file", line 1: warning: S has S::~S() but no constructor
"file", line 2: not implemented: local static s has S::~S() but no
constructor
(add S:: S())
```

- `lvalue` *op* `too complicated`

  This message should not be produced.

- needs temporary variable to evaluate argument initializer

  A default argument requires a temporary variable.

```
void f() {
      int g(const int& = 5);
}
```

```
"file", line 2: not implemented: needs temporary variable to
evaluate argument  initializer
```

- nested class *type* as parameter type to template class *name*

  A nested class is used as the actual parameter for a template class instantiation:

```
template <class T> struct S;

struct outer {
      struct inner {};
};

S<outer::inner> svar;
```

```
"file", line 7: not implemented: nested class outer::inner as
parameter type to template class S
```

- nested class *name* within nested class *name* within template class *name*

  Classes may only be nested directly within template classes, classes within nested classes within template classes are not implemented:

```
template <class T> class S {
class nest1 {
      class nest2 {/*...*/};
      };
};
```

```
"file", line 3: not implemented: nested class S::nest1::nest2
within nested class S::nest1 within template class S
```

- `nested depth class beyond 9 unsupported`

  Classes are nested more than nine levels deep.

```
struct S1 {
 struct S2 {
  struct S3 {
   struct S4 {
    struct S5 {
     struct S6 {
      struct S7 {
       struct S8 {
        struct S9 {
         struct S10 { enum { e }; };
};};};};};};};};};;
```

```
"file", line 20: not implemented: nested depth class beyond 9
unsupported
```

- `non-trivial declaration in switch statement`

  A "non-trivial" declaration appears within a switch statement. Such a declaration might declare an object of reference type, a static object, a `const` object, an object of a class type with constructor or destructor, an object with an initializer list, or an object initialized with a string literal.

```
void f(int i) {
        switch (i) {
        default:
                int& j = i;
        }
}
```

```
"file", line 2: not implemented: non-trivial declaration in switch
statement
(try enclosing it in a block)
```

  Since it is illegal to jump past a declaration with an explicit or implicit initializer unless the declaration is in an inner block that is not entered, most declarations in `switch` statements and not contained in inner blocks will be errors.

- `out-of-line definition of member function of class nested within template class`

  The member functions of a class nested within a template function must be defined within the definition of the nested class.

```
template <class t> struct x {
    struct y { void foo(); };
    // ...
};

template <class t>
    void x<t>::y::foo(){}
```

```
"file", line 7: not implemented: out-of-line definition of member
function of class  nested within template class ( x::y:: foo())
```

- `overly complex` *op* `of` *op*

  This message should not be produced.

- `parameter expression of type float, double or long double`

  A template taking a non-type argument is declared taking a float, double or long double argument:

```
template <double d> struct S { /*...*/};
```

```
"file", line 1: not implemented: parameter expression of type
float, double, or long double
```

- `postfix template function operator ++(): please make a class member function`

  The postfix implementation of a template increment or decrement operator must be a member function.

```
template <class t> struct x {
      int operator++(int); // ok
};

template <class t>
     int operator++(x<t>&,int); // sorry

x<int> xi;
```

```
"file", "", line 6: not implemented: postfix template function
operator ++(): please make a class member function
```

- `pointer to member function` *type* too complicated

  This message should not be produced.

- `public specification of overloaded` *function*

  The base class member in an access declaration refers to an overloaded function. A similar message is issued for `private` and `protected` access declarations.

```
struct B { int f(); int f(int); };
class D : private B {
public:
        B::f;
};
```

```
"file", line 2: not implemented: public specification of overloaded
B::f()
```

- `reuse of formal template parameter` *name*

  A template formal parameter name is reused within the template declaration:

```
template <class T> struct S {
      int T;
};
```

```
"file", line 2: not implemented: reuse of formal template parameter
T
```

- `specialized template` *name* `not at global scope`

  A specialized template is declared at other than global scope:

```
template <class T> struct S {
      T var;
};

void f() {
     struct S <int > {
           int var;
     };
};
```

```
"file", line 6: not implemented: specialized template S not at
global scope
```

- `static member anonymous union`

  A static class member is declared as an anonymous union.

```
class C {
      static union {
            int i;
            double d;
      } ;
};
```

```
"file, line 5: not implemented: static member anonymous union
```

- `struct` *name* `member` *name*

  This message should not be produced.

- template function actuals too complicated (please simplify)

```
#include <iostream.h>

template <class i> struct x { x(); };

template <class t>
ostream& operator<<(ostream &os, x<t>&) { return os; }

x<int> z;

main() {
/*
 * ok: simplified invocation of actual template function:
 * cout << "hello"; cout << z << endl;
 */

// generates sorry message: actuals too complicated
cout << "hello" << z << endl;
}
```

```
"file", line 17: not implemented: template function operator <<():
actuals too complicated (please simplify)
```

- template function instantiated with local class *name*

```
template <class T> int f(T);

f2() {
     struct local {/*...*/};
     local lvar;
     f(lvar);
}
```

```
"file", line 6: not implemented: template function f() instantiated
with local class local
```

- temporary of class *name* with destructor needed in *expr* expression

An expression containing a `?:`, `||`, or `&&` operator requires a temporary object of a class that has a destructor.

```
struct S { S(int); (apS(); };
S f(int i) {
      return i ? S(1) : S(2) ;
}
```

```
"file", line 3: not implemented: temporary of class S with
destructor needed in ?: expression
```

- too few initializers for *name*

  The initializer list for an array of class objects has fewer initializers than the number of elements in the array.

```
struct S { S(int); S(); };
S a[2] = {1};
```

```
"file", line 2: not implemented: too few initializers for ::a
```

- *type1* assigned to *type2* (too complicated)

  A pointer is initialized or assigned with an expression whose type is too complicated.

```
struct S1 {};
struct S2 { int i; };
struct S3 : S1, S2 {};
int S3::*pmi = &S2::i;
```

```
"file", line 4: not implemented: int S2::* assigned to int S3::*
(too complicated)
```

- use of *member* with formal template parameter

An attempt to use a member of a formal parameter type, such as `T::type`, is not currently supported. For example,

```
template <class T> class U
{
     typedef T TU;
     // ...
};

template <class Type> class V
{
     Type::TU t;
     // ...
};
```

```
"file", line 9: not implemented: use of Type::TU with formal
template type parameter
"file", line 9: cannot recover from earlier errors
```

- `visibility declaration for conversion operator`

  An access declaration is specified for a conversion operator.

```
struct B { operator int(); };
class D : private B {
public:
B::operator int;
};
```

```
"file", line 1: not implemented: visibility declaration for
conversion operator
```

- `volatile functions`

  A member function is specified as `volatile`.

```
struct S {
     int f() volatile;
};
```

```
"file", line 2: not implemented: volatile functions
```

- `wide character constant`
- `wide character string`

  A wide character constant or a wide character string is used.

```
int wc = L'ab';
char *ws = L"abcd";
```

```
"file", line 1: not implemented: wide character constant
"file", line 2: not implemented: wide character string
```

Index

Symbols

Numerics

A

B

C

D

E

F

I

K

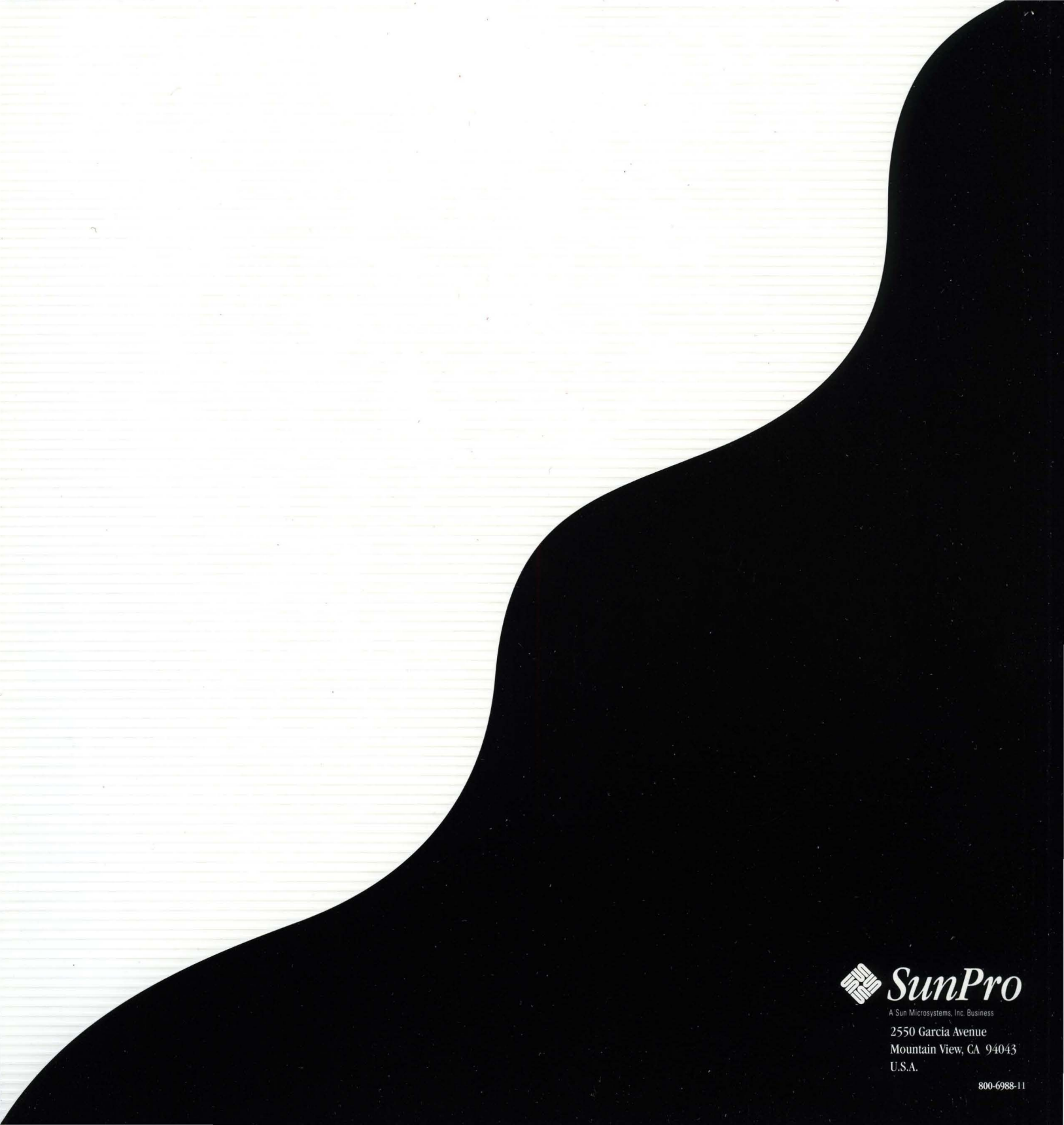
L

M

N

O

P



SunPro
A Sun Microsystems, Inc. Business
2550 Garcia Avenue
Mountain View, CA 94043
U.S.A.

800-6988-11