The C++ standard has several problems: it is inconsistent, some of its examples are ill-formed C++, some specifications are completely missing. Here is a list of problems that we have found.

[C/C++] means that the problem is the same in C and in C++.
[std] means that it is about standardize more than about C++
NRAD not really a defect

**Core language**

**Name lookup**

A section refers to itself [NRAD]
In 3.4.1, there is a reference to 3.4.1. It appears to be correct but it is a bit strange.

Lookup rules are inconsistent [NRAD]
The well-known rule that overloading does not work across scopes is wrong. Rules for name lookup in namespaces and classes rmembers are inconsistent.

**Memory model**

The draft is contradicting wrt manipulations of memory with char*
The intent is to allow some operations with char* (mainly copy and pointer arithmetic), but the draft is not entirely clear. reinterpret_cast to char* does not have a well-defined meaning; we are not sure about char* arithmetic.

Ref: 3.9/2

The notion of dynamic type is not well-defined
Clarify the notion of dynamic type: difficult for POD. Also say what happens with a union. The semantics of unions are not clear

A reference is rebindable [NRAD]
This is surprising and unnatural. This can also cause subtle optimizer bugs.

Example:

```cpp
struct T {
    int& ri;
    T (int& r) : ri (r) { }
};
void bar (T*);
void foo () {
    int i;
    T x (i);
    x.ri = 3; // the optimizer understands that
```
// this is really i = 3
bar (&x);
x.ri = 4; // optimizer assumes that this writes
// to i, but this is incorrect
}
int gi;
void bar (T* p)
{
    p->~T ();
    new (p) T (gi);
}

If we replace T& with T* const in the example then undefined behavior result and the optimizer is correct.
Proposal: make T& equivalent to T* const by extending the scope of 3.8/9 to references
Ref: 3.8/9

Expressions

Special members functions

Problem with assignment
It is unspecified for virtual base classes if operator= is called only once by derived class operator= this should be implementation defined.

Either the implementor makes the effort to call assignment operator only once, using some flag to know when to call them (as in a constructor), or he does not care and call them in all derived assignment operator (resulting in more than one call for virtual base classes). This is a user friendly/ easily implemented tradeoff and should be documented.
Proposal: change unspecified to implementation defined
Ref: 12.8 p 12-22 / 13

Assignment
12.8/10 Copying class objects [class.copy]

Otherwise, the implicitly declared copy constructor will have the form X& X::operator=(X&)

Description of copy ctor is contradicting ***
Is a copy ctor a T::T (cv T&) according to 12.8/2 or just T::T (T &) or T::T (const T&) according to 12.1/10 ?

There in Oct 96, and Jun 97, not checked with Nov 97..

Assignment to rvalues
We were previously under the impression that assignment to rvalue was forbidden, but we do not find this anywhere.
Issues to consider:
- is it an accident ?
- how do we define Rvalue vs Lvalue in a simple way (in textbooks) ? try to explain that. They see if your explanation is right or wrong
- consistency with other member functions

Exceptions

Static checking of exceptions specifications
The specification of checking of exceptions specifications is silly; this is especially visible in C++PL 14.6.1 p 377 since two contradicting statements follow each other:
[function pointer assignment ] // error: g() less restrictive than pf3

An exception-specification is not part of the type of a function

Templates

Non-type parameters
Non-type template parameters should be compile-time constants, but example 14.1/8 uses a variable.
**Library**

**Introduction**

Members added by the implementation[std]

In 17.3.4.4/2 vs 17.3.4.7/0 there is a hole; an implementation could add virtual members a base class and break user derived classes.

Example:

```cpp
// implementation code:
struct _Base { // _Base is in the implementer namespace
    virtual void foo () ;
};
class vector : _Base // deriving from a class is allowed
    { ... };

// user code:
class vector_checking : public vector
    { void foo (); // don't want to override _Base::foo () as the
        // user doesn't know about _Base::foo ()
    };
```

Proposal: clarify the wording to make the example illegal

**STL**

```cpp
vector<bool> ***
```

vector<bool> is not a container as its reference and pointer types are not references and pointers. This is an old problem but it is still there.

Also it forces everyone to have a space optimization instead of a speed one.

Proposal: rename it bit_vector. vector<bool> is a normal vector with no space optimization. Stop pretending bit_vector is a container.

**Insert inconsistent definition**

insert (iterator, const value_type&) is defined both on sequences and on set, with unrelated semantics: insert here (in sequences), and insert with hint (in associative containers). They should have different names (B.S. says: do not abuse overloading).

**Input iterator requirements are badly written**

For example, they cannot even be satisfied by pointers!

Table 63 p 24-3 is broken

Not in CD1 but in CD2.

Proposal: add for *x++: «To call the copy constructor for the type T is allowed but not required.»

**reverse_iterator comparisons completely wrong *****

Bug The <, >, <=, >= comparison operator are wrong: they return the opposite of what they should.

Proposal: simply reverse them

Ref: 24.4.1.3.13 to 17; p 24-14

Note: same problem in CD2, these were not even defined in CD1

SGI STL code is correct; this problem is known since the Morristown meeting but there it was too late

**Insert iterators/ostream_iterators overconstrained**

Overspecified For an insert iterator it, the expression *it is required to return a reference to it. This is a simple possible implementation, but as the SGI STL documentation says, not the only one, and the user should not assume that this is the case.

Proposal: remove the requirement, and let the implementation return any proxy class. This proxy class can still be the iterator itself.

Ref: 24.4.2 p 24-15 and 24.5.4 p 24-25

**No way to free storage for vector and deque**

reserve can not free storage, unlike string::reserve.

Proposal: reword reserve
Bug in insert range in associative containers
Table 7 of Containers say that a.insert(i,j) is linear if [i, j) is ordered.
It seems impossible to implement, as it means that if [i, j) = [x], insert in an associative container is O(1) !
Proposal: N + log (size()) if [i,j) is sorted according to value_comp()

set::iterator is required to be modifiable
set::iterator is described as implementation-defined with a reference to the container requirement; the
container requirement says that const_iterator is an iterator pointing to const T and iterator an iterator
pointing to T

Overloading
In 13.1 Overloadable declarations   [over.load]/3
void f ( int i = 88, int j); // OK: redeclaration of f(int, int)
This is incorrect C++.

String library
Description of operator[] is unclear [std]
It is not clear that undefined behavior applies when pos == size () for the non const version.
Proposal: rewrite as: Otherwise, if pos > size () or pos == size () and the non-const version is used,
then the behavior is undefined.
Ref: 21.3.4 p 21-16

Iostream
fstream ctors take a const char* instead of string (extension ?)
fstream ctors can't take wchar_t (extension ?)
An extension to add a const wchar_t* to fstream would make the implementation non conforming.

Numeric library
A private member is implementation defined
This is the only place in the whole standard where the implementation has to document something private.
Proposal: change implementation defined to unspecified
Ref: 26.3.5 - p 272

valarray constructor is strange
The order of the arguments is (elem, size) instead of the normal (size, elem) in the rest of the library. Since
elem often has an integral or floating point type, both types are convertible to each other and reversing them
leads to a well formed program.
Proposal: Inverting the arguments could silently break programs. Introduce the two signatures (const T&,
size_t) and (size_t, const T&), but make the one we do not want private so errors result in a
diagnosed access violation. This technique can also be applied to STL containers.
Ref: 26.3.2.1/ 2

Exceptions
exception::what()
The lifetime of the return value of exception::what() is left unspecified. This issue has implications with
exception safety of exception handling: some exceptions should not throw bad_alloc. This is not a trivial
issue.