The Dark Side of C++

Felix von Leitner
CCC Berlin
felix-cpp@fefe.de

August 2007

Abstract

Most of the perceived benefits of C++, when viewed from a different vantage point, turn out to have or even be downsides.

Central Complaints

1. new bug classes
2. hard to write
3. hard to read

Warm-Up

```
readply.cpp:109: conversion from ‘_List_iterator<list<basic_string<c
har,string_char_traits<char>,__default_alloc_template<true,0> >,allo
cator<basic_string<char,string_char_traits<char>,__default_alloc_tem
plate<true,0> > > >,const list<basic_string<char,string_char_traits<
char>,__default_alloc_template<true,0> >,allocator<basic_string<char
,string_char_traits<char>,__default_alloc_template<true,0> > > > &,c
onst list<basic_string<char,string_char_traits<char>,__default_alloc
_template<true,0> >,allocator<basic_string<char,string_char_traits<c
har>,__default_alloc_template<true,0> > > > *>’ to non-scalar type ‘
list<basic_string<char,string_char_traits<char>,__default_alloc_temp
late<true,0> >,allocator<basic_string<char,string_char_traits<char>,
__default_alloc_template<true,0> > > >’ requested
```

Actually...

```
#include <list>
#include <string>
#include <algorithm>

using namespace std;

void foo(list<string>& x) {
  sort(x.begin(),x.end());  // wrong iterator type
}
```

Actually...

```
/usr/lib64/gcc/x86_64-unknown-linux-gnu/4.2.1/../../../../include/c++/4.2.1/bits/stl_algo.h: In func\
tion 'void std::sort(_RandomAccessIterator, _RandomAccessIterator) [with _RandomAccessIterator = std\
::_List_iterator<std::basic_string<char, std::char_traits<char>, std::allocator<char> > >]':
big-error.C:8:   instantiated from here
/usr/lib64/gcc/x86_64-unknown-linux-gnu/4.2.1/../../../../include/c++/4.2.1/bits/stl_algo.h:2829: er\
ror: no match for 'operator-' in '__last - __first'
/usr/lib64/gcc/x86_64-unknown-linux-gnu/4.2.1/../../../../include/c++/4.2.1/bits/stl_algo.h: In func\
tion 'void std::__final_insertion_sort(_RandomAccessIterator, _RandomAccessIterator) [with _RandomAc\
cessIterator = std::_List_iterator<std::basic_string<char, std::char_traits<char>, std::allocator<ch\
ar> > >]':
/usr/lib64/gcc/x86_64-unknown-linux-gnu/4.2.1/../../../../include/c++/4.2.1/bits/stl_algo.h:2831:   \
instantiated from 'void std::sort(_RandomAccessIterator, _RandomAccessIterator) [with _RandomAccessI\
terator = std::_List_iterator<std::basic_string<char, std::char_traits<char>, std::allocator<char> > >]'
big-error.C:8:   instantiated from here
/usr/lib64/gcc/x86_64-unknown-linux-gnu/4.2.1/../../../../include/c++/4.2.1/bits/stl_algo.h:2436: er\
ror: no match for 'operator-' in '__last - __first'
/usr/lib64/gcc/x86_64-unknown-linux-gnu/4.2.1/../../../../include/c++/4.2.1/bits/stl_algo.h:2438: er\
ror: no match for 'operator+' in '__first + 16'
/usr/lib64/gcc/x86_64-unknown-linux-gnu/4.2.1/../../../../include/c++/4.2.1/bits/stl_algo.h:2439: er\
ror: no match for 'operator+' in '__first + 16'
/usr/lib64/gcc/x86_64-unknown-linux-gnu/4.2.1/../../../../include/c++/4.2.1/bits/stl_algo.h: In func\
tion 'void std::__insertion_sort(_RandomAccessIterator, _RandomAccessIterator) [with _RandomAccessIt\
erator = std::_List_iterator<std::basic_string<char, std::char_traits<char>, std::allocator<char> > >]':
/usr/lib64/gcc/x86_64-unknown-linux-gnu/4.2.1/../../../../include/c++/4.2.1/bits/stl_algo.h:2442:   \
instantiated from 'void std::__final_insertion_sort(_RandomAccessIterator, _RandomAccessIterator) [w\
ith _RandomAccessIterator = std::_List_iterator<std::basic_string<char, std::char_traits<char>, std:\
:allocator<char> > >]'
/usr/lib64/gcc/x86_64-unknown-linux-gnu/4.2.1/../../../../include/c++/4.2.1/bits/stl_algo.h:2831:   \
instantiated from 'void std::sort(_RandomAccessIterator, _RandomAccessIterator) [with _RandomAccessI\
terator = std::_List_iterator<std::basic_string<char, std::char_traits<char>, std::allocator<char> > >]'
big-error.C:8:   instantiated from here
/usr/lib64/gcc/x86_64-unknown-linux-gnu/4.2.1/../../../../include/c++/4.2.1/bits/stl_algo.h:2352: er\
ror: no match for 'operator+' in '__first + 1'
/usr/lib64/gcc/x86_64-unknown-linux-gnu/4.2.1/../../../../include/c++/4.2.1/bits/stl_algo.h:2442:   \
instantiated from 'void std::__final_insertion_sort(_RandomAccessIterator, _RandomAccessIterator) [w\
ith _RandomAccessIterator = std::_List_iterator<std::basic_string<char, std::char_traits<char>, std:\
:allocator<char> > >]'
/usr/lib64/gcc/x86_64-unknown-linux-gnu/4.2.1/../../../../include/c++/4.2.1/bits/stl_algo.h:2831:   \
instantiated from 'void std::sort(_RandomAccessIterator, _RandomAccessIterator) [with _RandomAccessI\
terator = std::_List_iterator<std::basic_string<char, std::char_traits<char>, std::allocator<char> > >]'
big-error.C:8:   instantiated from here
/usr/lib64/gcc/x86_64-unknown-linux-gnu/4.2.1/../../../../include/c++/4.2.1/bits/stl_algo.h:2358: er\
ror: no match for 'operator+' in '__i + 1'
```

Why I Learned C++

- OOP with Inline Member Functions
- (Operator) Overloading
- Templates

  (STL didn’t exist yet)

Ever-Changing Standard

- Wrote commercial C++ app in 1997
- Then C++ changed (life time of i in ″for (int i=0; ...)″)
- Then C++ changed again (use iostream instead of iostream.h)
- Then C++ changed AGAIN (namespaces)

Useless maintenance work. Would have been less trouble in C.

Ever-Changing Styles

Old and busted:

```
for (int i = 0; i < n; i++)
```

New hotness:

```
for (int i(0); i != n; ++i)
```

Optimizing in the Wrong Direction

- C++ makes using good libraries very nice
- C++ makes writing good libraries almost impossible
- Reading libraries is even harder

Reading vs Writing

- Code must be easy to follow, not easy to write
- Code is written only once, but read many more times
- More context needed means harder to read

C++ is hard to parse

Not just for humans, too:

```
template<class T> struct Loop { Loop<T*> operator->(); };
Loop<int> i, j = i->hooray;
```

This sends the compiler into an endless loop until swap exhaustion.

C++ is hard to parse

```
struct a{typedef int foo;};struct a1:a{};struct a2:a{};
#define X(b,a) struct a##1:b##1,b##2{};struct a##2:b##1,b##2{};
X(a,b)X(b,c)X(c,d)X(d,e)X(e,f)X(f,g)X(g,h)X(h,i)X(i,j)X(j,k)X(k,l)
X(l,m)X(m,n) n1::foo main(){}
```

In each iteration, the compiler must check whether the types of `foo` on both sides of the multiple inheritance match.

g++ used to eat a gig of RAM and a couple CPU hours. icc still does.

C++ is hard to parse

- `string a("blah");` instantiates a string and assigns "blah", but `string a();` declares a local function returning a string.
- `a && b` evaluates a, and if it's true, it also evaluates b.
  Unless someone declared `operator&&`, then both are evaluated.
- `(foo) + a*b;` becomes +*(foo,* **(a,b))*.
  Unless foo is a type, then it's **(typecast(a,foo),b)*.
- `a, b` means a is evaluated before b.
  Unless someone declared `operator,`, then it's not guaranteed.

C++ is hard to write

`cin` has a type conversion to `void*`

So you can do `while (cin) { ...`

Why `void*` and not `bool`?

Because then `cin << 3` would not be a compile time error.

It would convert `cin` to `bool`, convert that to `int` and left-shift it by `3`.

C++ is hard to write

- Can't throw exceptions in destructors, shouldn't in constructors
- Initializers done in order of declaration of fields in class, not written order
- `auto_ptr` is useless
- Iterators don't know anything about the container, can't detect errors
- For a `vector`, `at()` does bounds checking, but `operator[]` doesn't
- Can't call virtual member functions from constructor or destructor

Throwing exceptions from con/destructors

- `delete[]` would leak memory if object 2 of 10 threw an exception
- Exceptions in constructor don't unwind the constructor itself
- Does not even clean up local variables!
- Must do own cleanup
- OTOH: no other way to return failure, constructors are `void`

Initializer Order

```
template<typename T> class Array {
private:
  T* data;
  size_t allocated, used;
public:
  Array(int n) : allocated(n), used(0), data(new T[allocated]) {}
};
```

Initialization order is declared in the class itself, not in the constructor.

`data` is allocated with the wrong size here.

auto_ptr sucks

- Supposed to provide ”poor man’s garbage collection”
- If you copy it, only the copy retains ownership
- The original silently breaks
  - no explicit assignments
  - no by-value passing into functions
  - breaks when put into STL containers
- The next version of the standard has better alternatives

Iterators and their Containers

- Iterators are like dumb pointers
- Don’t know anything about the container
- Silent breakage if you do `a.remove(b.begin())`
- Iterator to removed element
- Iterator to stale element (`resized` vector, or balanced tree after rebalance)
- Iterator into destructed container

Bounds Checking

- For a `vector`, `at()` does bounds checking
- `operator[]` can, but does not have to
- The one from gcc doesn’t

Virtual functions in con/destructor

- In the constructor, the vtable is not properly initialized
- Usually, function pointers in vtable still point to virtual member of base class
- … which can be pure virtual
- In the destructor of the base class, vtable points to base class

C++ is hard to read

```
Type a = b;
a.value = 23;
```

Looks like `Type` is some kind of struct or class and `b` is left alone.

But what if:

```
typedef OtherType& Type;
```

Need to look at the types, too.

C++ is hard to read

```
baz = foo->bar(3);
```

What does this code do? Looks easy enough, right?

C++ is hard to read

```
baz = foo->bar(3);
```

`baz` could be an int.

`foo` could be a pointer to some struct.

`bar` could be a function pointer.

C++ is hard to read

```
baz = foo->bar(3);
```

`baz` could be a **reference** to int (suddenly we overwrite something elsewhere).

`foo` could be a class with its own `operator->` (cascading smart pointers!)

`bar` could be a member function

`baz` could even be a reference to a `typeof(foo)` and alias `foo`.

C++ is hard to read

```
baz = foo->bar(3);
```

Which `baz` is it? `::baz` or `whatever::baz`?

Can't say from this line, have to look for

```
using namespace whatever;
```

What if the code is using more than one namespace?

Those declarations could be in some included header file.

C++ is hard to read

```
baz = foo->bar(3);
```

There could be templates involved.

Maybe more than one template?

Maybe there are specializations somewhere that change the semantics?

(That is particularly bad when you use STL primitives and part of their operation gets overwritten by some specialization somewhere; think about `sort` which depends on `operator<`)

C++ is hard to read

```
baz = foo->bar(3);
```

If `foo->` references some class, then `bar` could be a virtual function. Which one is actually called?

If we have inheritance, all functions of the same name (even with different signatures) are hidden by a function of that name in the derived class.

C++ is hard to read

```
baz = foo->bar(3);
```

There could be overloaded versions of `bar`. We would have to check which ones accept an `int` as argument.

There could be overloaded versions of `bar` with default arguments.

There could be ones that don't accept `int`, but where a type conversion to `int` exists (enum, bool, custom, ...).

C++ is hard to read

```
baz = foo->bar(3);
```

There could be implicit type conversions. Maybe `foo->bar(3)` returns a `fnord`. Now we have to check if and how the type is getting converted to the type of `baz`.

The assignment operator could be overloaded. `->` could be overloaded. `()` could be overloaded. These implementations could use other overloaded operators that we would have to track down.

Depending on how well the types match, maybe a temp object is generated. Then we would have to look at the constructors and destructors, too.

C++ is hard to read

```
baz = foo->bar(3);
```

Obviously, there could be preprocessor tricks.

Those are in C, too, and I like C, so I’m not bashing them here.

I do agree that they need some bashing.

C++ makes auditing painful

A typical problem for a source code auditor is:

- I just found this buffer overflow
- How do I trigger this?

This sucks even for C, but for C++ it depends on run-time state like the vtables. A typical C++ application takes more than twice as long to audit as a typical C application.

C++ makes auditing painful

References make auditing much harder than it has to be.

When looking for integer overflows, you look for allocations, then you go up looking for checks and places where the involved integers are written.

With C++, you don't see the writes, because it says

```
some_func(whatever,the_int,SOME_FLAG);
```

and it can still write `the_int` if it's a reference. In C, it would be `&the_int`, which is easily recognizable.

C++ is too hard

C++ is like a C compiler that leaves the error messages to the assembler.

When you feed it syntactically bad code, it will generate syntactically bad assembler code.

Almost nobody actually understands the error messages.

You get an error message? You start fudging the code until it compiles.

C++ is too powerful

- Boost.Lambda: `for_each(a.begin(), a.end(), std::cout << _1 << ' ')`
- C++ Expression Templates
- `Vector a,b,v; v = a /vectorProduct/ b;`
- Most people can't handle all that flexibility
  - Code leaks resources when someone throws an exception
  - Have to provide assignment operator, but fail if someone does `a=a;`
  - Get hurt by stumbling blocks like `auto_ptr`

C++ has some horrible warts

- `assert(s[s.size()] == 0);` works if `s` is a `const std::string`, but is undefined if it is not `const`.
- To overload `++a`, you declare an `operator++(yourtype&)`. To overload `a++`, you declare an `operator++(yourtype&,int dummy)`.
- Using custom allocators for STL containers precludes interoperating with non-custom-allocator containers.
- `operator[]` adds a member to a `map` if it does not already exist.
- For templates, you have to substitute "> >" for ">>"

Other nitpicks

- When disassembling, you can detect C++ code by the tons of do-nothing wrappers
- Linkers remove unreferenced code

  … but virtual function tables do reference the virtual functions.

  So the linker includes all overwritten virtual functions from base classes.
- Can't tell what exceptions a library can throw

New Bug Classes

- exception safety (resource leaks, deadlocks)
- `delete` vs `delete[]` (code execution)
- integer overflow in `operator new[]` (code execution, fixed in MSVC)
- local static initialization not thread-safe
- many horrible warts
  - expired iterators (code execution)
  - pointer arithmetic with base classes (memory corruption)

Exception Safety

```
char* read_first_n_bytes_from_file(const char* file,size_t n) {
  int handle=open(file,O_RDONLY);
  pthread_mutex_lock(&global_mutex);
  char* buf=new char[n];
  readfile(handle,buf,n);
  pthread_mutex_unlock(&global_mutex);
  close(handle);
  return buf;
}
```

If `new[]` throws an exception, this leaks a file handle and deadlocks the next caller.

delete vs delete[]

```
void foo(int* x) {
  delete[] x;
}
```

This will just call `operator delete`, which calls `free()`.

delete vs delete[]

```
void bar(class foo* x) {
  delete[] x;
}
```

This also calls the destructors.

How does it know how many destructors to call?

delete vs delete[]

```
void bar(class foo* x) {
  delete[] x;
}

int n=((int*)x)[-1];
class foo* y=x[n-1];
while (y!=x) {
  y->~foo();
  --y;
}
delete x;
```

```
        testq   %rdi, %rdi
        je      .L6
        movq    -8(%rdi), %rax
        leaq    (%rdi,%rax,4), %rbx
        cmpq    %rbx, %rdi
        je      .L4
.L7:
        subq    $4, %rbx
        movq    %rbx, %rdi
        call    _ZN3fooD1Ev
        cmpq    %rbx, %rbp
        jne     .L7
.L4:
        addq    $8, %rsp
        leaq    -8(%rbp), %rdi
        popq    %rbx
        popq    %rbp
        jmp     _ZdaPv
.L6:
```

delete vs delete[]

If you call `delete` when you should have called `delete[]`, the pointer will be off by sizeof(int), leading to heap corruption and possibly code execution.

If you call `delete[]` when you should have called `delete`, some random destructors will be called on garbage data, probably leading to code execution.

Integer overflow in operator new[]

```
wchar_t mystrdup(const char* x,size_t n) {
  wchar_t* y=new wchar_t[n];
  for (size_t i=0; i<n; ++i)
    y[i]=x[i];
  return y;
}
```

What if n is 0x80000000? The compiler multiplies by `sizeof(wchar_t)` and passes the result to `operator new[]`, which basically calls `malloc()`.

Straight integer overflow with potential code execution.

Integer overflow in operator new[]

MSVC 2005 adds code to check for int overflows (my fault, sorry).

The code still calls `new[]`, but then passes -1, i.e. 0xffffffff. Normally, `new[]` calls malloc, which then fails. But what if you have overloaded `new[]` so that it adds a small header?

Local static initialization not thread-safe

```
int bar() {
  static struct foo bla;
  return bla.member;
}
```

```
        cmpb    $0, _ZGVZ3barvE3bla
        je      .L5
        movl    _ZZ3barvE3bla, %eax
        addl    $12, %esp
        ret
.L5:
        [call constructor]
        movb    $1, _ZGVZ3barvE3bla
        [handle exceptions]
        movl    _ZZ3barvE3bla, %eax
        ret
```

gcc 4 calls a guard function in libstdc++ that actually acquires a (global, static) mutex.

Pointer arithmetic with base classes

```
struct bar { int member; };
struct foo : public bar { int member2; };

void frobnicate(struct bar* x) {
  x[0] -= x[1];    // x[1] adds sizeof(bar) here, not sizeof(foo)
}

extern struct foo* array;

int ohnoooo() {
  frobnicate(array);
}
```

Gratuitous Bjarne Quote at the end

Whole program analysis (WPA) can be used to eliminate unused virtual function tables and RTTI data. Such analysis is particularly suitable for relatively small programs that do not use dynamic linking.

No shit, Sherlock.

It will be ready right after Duke Nukem Forever runs on the Hurd.

PS: Thanks to helpers and contributors, in particular Stefan Reuther.