Fornux C++ Superset

Manual

Table of Contents

Introduction

Why another memory manager? Because to this day memory management can be subdivided in two categories:

- garbage collection;
- reference counting.

And we will see below that neither of them is objective enough to cover all use cases.

In one hand the garbage collectors do not offer deterministic destruction of the objects which creates this unpredictable behavior that is unpleasant to see on a front end user-interface experience for example.

On the other hand the reference counting is deterministic but do not offer any way to detect cyclic references. So no clean solution exists to resolve this problem up until now.

We will see below a description of the aforementioned items and the solution proposed by this library to solve this.

Garbage collection

Garbage collection is a technique where memory blocks are collected and later deallocated when they are found to be unreferenced by any other object.

Garbage collection is used by many popular languages including:

- ML
- Haskell
- APL
- Lisp
- Ruby
- Java
- .NET (C#)?
- Smalltalk
- ECMAScript
- Sather

Garbage Collector Advantages are:

- Very fast allocation and deallocation timing.
- Dangling pointer bugs.
- Double free bugs.

- Certain kinds of memory leaks, leading over time to memory exhaustion.
- Possible to override system allocators without modifying any code

Garbage Collector Disadvantage:

- Cannot reclaim memory or invoke finalizers / destructors block immediately.
- Cannot reclaim all unreachable objects.
- Doesn't know which registers will be referenced.

An example of its usage is demonstrated here with the C/C++ implementation of the popular Hans Boehm garbage collector:

```
#include <gc.h>



int main()

{

    int * p = (int *) GC_MALLOC(sizeof(int) * 100); // Allocate an array of 100
integers.



    return 0;

} // non-deterministic deallocation of the array.
```

Unfortunately this technique simply postpones the deallocation of the unreferenced objects to later freeze the entire application, on a single CPU system, and to collect them using various tracing algorithms. This may be unacceptable for real-time applications or device driver implementation, for example.

For solar or battery-powered devices, the power consumed by garbage collection might also be significant.

Reference counting

Reference counting is a different approach where objects pointed to are aware of the number of times they are referenced.

This means a counter within the object is incremented or decremented according to the number of smart pointers that are referencing or dereferencing it.

For example, using boost::shared_ptr (or the std:: version):

```
#include <boost/smart_ptr/shared_ptr.hpp>
```

```
#include <boost/smart_ptr/weak_ptr.hpp>

using boost::shared_ptr;

using boost::weak_ptr;


#include <boost/make_shared.hpp>

using boost::make_shared;

struct S

{

  shared_ptr<S> q;

  ~S()

  { //

    std::cout << "~S()" << std::endl; // ~S() output when S is destructed.

  }

};

shared_ptr<int> p = make_shared<int>(11);

shared_ptr<int> q = p;


p.reset();


std::cout << *q << std::endl; // Outputs 11.
```

The main drawback is a lost in performance as compared to garbage collection because of the extra time required to manage the counter every time the pointer is reassigned or dereferenced.

Reference counting can also leave a group of blocks of memory referencing each other (called "cyclic") unnoticed and therefore never freed by the application.

A cyclic set is shown here:

```
struct S

{

  shared_ptr<S> q;

  ~S()

  { //
```

```
    std::cout << "~S()" << std::endl; // ~S() output when S is destructed.
  }
};
shared_ptr<S> p = make_shared<S>();
p->q = p; // Create a cycle.


p.reset(); // Detach from the cycle.
```

The above example will never execute the call of the destructor of struct A because the cycle will never get deallocated. This is because the number of references will never reach 0.

A way to solve this problem is to isolate the location of the cycle, which is this case is already known, and use a weak_ptr to break the cycle:

```
struct W
{
  weak_ptr<W> q;
  ~W()
  { //
    std::cout << "~W()" << std::endl; // ~W() output when W is destructed.
  }
};
shared_ptr<W> p = make_shared<W>();
p->q = p; // Create a cycle.


p.reset(); // Detach from the cycle.
```

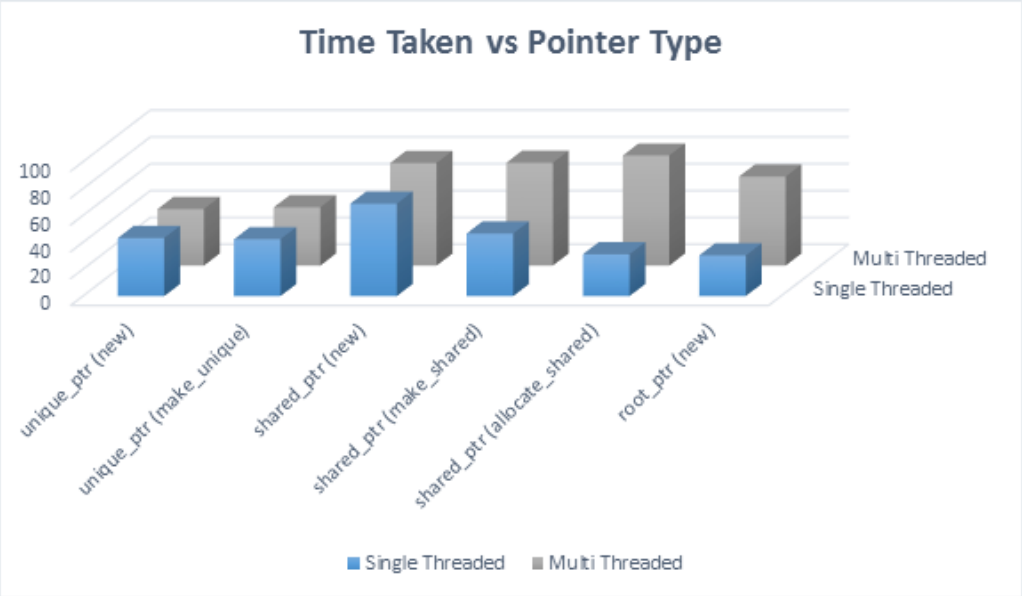
The main drawback of this approach is the need to explicitly find the cycle to later alter the code with respective weak_ptrs.

Root pointer

The solution we present here to solve these problems consists of altering the source code and completely replacing the memory management back-end using the patented "*root_ptr*" smart pointer.

That "*root_ptr*" is more compact than the standard "*shared_ptr*" and extends its functionality by merging pointers, iterators, pointer to arrays and pointer names to properly backtrace thrown exceptions.

Will there be a performance loss? According to the following graphic, the answer is no if we compare it to a "*shared_ptr*":



In other words, there is no need to any other smart pointer if "*root_ptr*" is used properly for new code.

Tutorial

C++ Superset can be launched by adding to your path the folder containing the "*fcxxss.sh*" script. This script will execute the necessary steps to convert the source code and then compile the resulting code.

Note that the precompiled headers and the resulting code are stored in the global folder "*/var/tmp/fcxxss*". So if compilation flags change then this folder will have to be deleted in order to refresh the precompiled header.

Pointers, iterators and arrays

The following first example shows how C++ Superset makes the distinction between a pointer, and an iterator to single objects and array of objects. This is one major flaw in the core C/C++ language that is properly fixed by C++ Superset.

It is also shown the usage of the array-style "*operator []*" on a pointer or iterator to single objects will thrown an exception that can be later captured by a "try ... catch" statement.

Lastly, pointers and iterators can be interchanged as desired.

The following example was compiled with the following command:

```
fcxxss.sh -DBOOST_DISABLE_THREADS memoryleak1.cpp
```

```
/**
    C++ Superset -- Example #1

    Outputs:

    1... 2...
    3... 4...
    1
    3
    A
    "non_array_pointer" is not a buffer
    #0 main in memoryleak1.cpp line 48
    #1 main in memoryleak1.cpp line 36

    1... 2...
    1... 2...

 */



#include <string.h>

#include <iostream>



using namespace std;
```

```
int main()
{
    // Handles iterators, pointers and arrays:
    char const * array_iterator = "1... 2...";
    char const * array_pointer = strdup("3... 4...");
    char const * non_array_pointer = new char{'A'}; // C++11-style initialization

    cout << array_iterator << endl;
    cout << array_pointer << endl;

    cout << array_iterator[0] << endl;
    cout << array_pointer[0] << endl;

    cout << * non_array_pointer << endl;

    try
    {
        cout << non_array_pointer[0] << endl;
    }
    catch (exception& e)
    {
        cout << e.what() << endl;
    }

    array_pointer = array_iterator; // 'array_pointer' is now an iterator

    cout << array_iterator << endl;
    cout << array_pointer << endl;

    return 0;
```

```
}
```

Cyclic memory leak management (C++)

The following example shows the long standing problem of cyclic references being solved by C++ Superset in a deterministic way, or predictably. This means the application will always execute exactly the same way every time it is launched.

You can see the difference with a normal compiler by compiling the following code and executing it. You'll see the memory usage of the application increasing linearly, which represents a memory leak. Such a memory leak can possibly bring an entire operating system down and creating a snowball effect with other components of the entire system.

After compiling the code with "*fcxxss.sh*", you will see the memory usage of the application being perfectly stable, without the need to debug or explicitly fix such memory leak because it is fixed implicitly.

The following example was compiled with the following command:

```
fcxxss.sh -DBOOST_DISABLE_THREADS memoryleak2.cpp
```

```
/**

    C++ Superset -- Example #2

    Outputs:

    Speed: 206185.15464 loops / s; Memory usage: 6764 kilobytes
    [...]
    Speed: 206185.36082 loops / s; Memory usage: 6764 kilobytes

 */



#include <stdlib.h>
#include <unistd.h>
```

```
#include <string.h>
#include <sys/resource.h>

#include <chrono>
#include <iostream>
#include <iomanip>



using namespace std;
using namespace std::chrono;



struct list_node
{
    list_node * p;
};

int main()
{
    // Proper cyclic memory leaks management:
    milliseconds before, after;

    before = duration_cast<milliseconds>(system_clock::now().time_since_epoch());

    for (int i = 0; i < 1000000; ++ i)
    {
        // cycle
        struct list_node * p = new list_node;
        p->p = new list_node;
        p->p->p = new list_node;
        p->p->p->p = p;
```

```
        // stats

        after = duration_cast<milliseconds>(system_clock::now().time_since_epoch());

        struct rusage r_usage;

        getrusage(RUSAGE_SELF, & r_usage);

        cout << "Speed: " << setprecision(11) << i * 1000.0 / (after - before).count()
<< " loops / s; Memory usage: " << r_usage.ru_maxrss << " kilobytes" << endl;

    }


    return 0;

}
```

Cyclic memory leak management (C)

C++ Superset can compile C source files as well but it will need to follow the C++ standards such as:

- explicit downcasting of pointer types;
- usage of "*nullptr*" over "*NULL*".

It has to be pointed out as well that calls to "*malloc*" will allocate the memory block and also initialize the object implicitly, just like a call to the "*operator new*" in C++.

The following example was compiled with the following command:

```
fcxxss.sh -DBOOST_DISABLE_THREADS memoryleak2.cpp
```

```
/**


    C++ Superset -- Example #2


    Outputs:


    Speed: 213036.178515 loops / s; Memory usage: 7196 kilobytes

    [...]
```

```
    Speed: 213036.210014 loops / s; Memory usage: 7196 kilobytes


 */



#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/time.h>

#include <sys/resource.h>



double time_diff(struct timeval x, struct timeval y);


struct list_node

{

    struct list_node * p;

};


int main()

{

    struct timeval before, after;


    gettimeofday(& before, (struct timezone *) nullptr);


    for (int i = 0; i < 1000000; ++ i)

    {

        // cycle

        struct list_node * p = (struct list_node *) malloc(sizeof(struct list_node));

        p->p = (struct list_node *) malloc(sizeof(struct list_node));

        p->p->p = (struct list_node *) malloc(sizeof(struct list_node));

        p->p->p->p = p;
```

```
        // stats

        gettimeofday(& after, (struct timezone *) nullptr);

        struct rusage r_usage;

        getrusage(RUSAGE_SELF, & r_usage);

        printf("Speed: %f loops / s; Memory usage: %ld kilobytes\n", i * 1000000.0 /
time_diff(before , after), r_usage.ru_maxrss);

    }



    return 0;

}



double time_diff(struct timeval x, struct timeval y)

{

    double x_ms , y_ms , diff;



    x_ms = (double)x.tv_sec*1000000 + (double)x.tv_usec;

    y_ms = (double)y.tv_sec*1000000 + (double)y.tv_usec;



    diff = (double)y_ms - (double)x_ms;



    return diff;

}
```

Efficient memory management

Some “*libstdc*” utilities where rewritten and optimized on the “*x86_64*” platform to act the way they should given the direction buffers are being copied by “*memcpy*”. In other words, the latter will work even if buffers are overlapping.

The following example was compiled with the following command:

```
fcxxss.sh -DBOOST_DISABLE_THREADS memoryleak3.cpp
```

```
/**

    C++ Superset -- Example #3

    Outputs:

    Test1... test2... this is a test
    Test1Test1... t.. this is a test
    Test1XXXXXXXXXX.. this is a test

 */



#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/resource.h>

#include <iostream>
#include <iomanip>



using namespace std;



int main()
{
    // Efficient memcpy() & memset():
    char * s = strdup("Test1... test2... this is a test");

    cout << s << endl;
```

```
    memcpy(s + 5, s, 10);

    cout << s << endl;


    memset(s + 5, 'X', 10);

    cout << s << endl;


    return 0;

}
```

STL containers

Pointers inside STL containers will be properly handled and destructed accordingly as you can see in the following example.

The following example was compiled with the following command:

```
fcxxss.sh -DBOOST_DISABLE_THREADS memoryleak4.cpp
```

```
/**


    C++ Superset -- Example #4


    Outputs:


    node 1

    Memory usage: 5724 kilobytes

    [...]

    node 1

    Memory usage: 5724 kilobytes


 */
```

```
#include <string.h>
#include <sys/resource.h>

#include <iostream>
#include <list>



using namespace std;



int main()
{
    // STL container friendly:
    for (int i = 0; i < 1000000; ++ i)
    {
        list<char *> l;
        l.push_back(strdup("node 1"));
        l.push_back(strdup("node 2"));
        l.push_back(l.front());

        cout << l.back() << endl;

        struct rusage r_usage;
        getrusage(RUSAGE_SELF, & r_usage);
        cout << "Memory usage: " << r_usage.ru_maxrss << " kilobytes" << endl;
    }

    return 0;
}
```

Lambda functions

Lambda functions, "*auto*" variable specifiers and all of the related access scopes are also properly handled.

The following example was compiled with the following command:

```
fcxxss.sh -DBOOST_DISABLE_THREADS memoryleak5.cpp
```

```
/**

    C++ Superset -- Example #5

    Outputs:

    void foo(boost::node_proxy &, int): 10
    void lambda::foo(boost::node_proxy &, int): 21
    void lambda::foo(boost::node_proxy &, int): 31

 */



#include <iostream>



using namespace std;



struct lambda
{
    void foo(int a)
    {
        int i = 1;
```

```
        auto n = i;
        auto p = & n;

        auto f = [this, i](int a) { return i + a; };

        cout << __PRETTY_FUNCTION__ << ": " << f(a) << endl;
    }
};

void foo(int a)
{
    cout << __PRETTY_FUNCTION__ << ": " << a << endl;
}

int main()
{
    // Lambda functions tests:
    lambda n;
    lambda m(n);

    foo(10);
    n.foo(20);
    m.foo(30);

    return 0;
}
```

Detection of buffer overflows

Buffer overflows are detected and will throw an exception if an iterator of an allocated heap object goes above the limits of such object.

Note that objects not allocated on the heap which are consequently iterated to will not throw any exception.

The following example was compiled with the following command:

```
fcxxss.sh -DBOOST_DISABLE_THREADS memoryleak6.cpp
```

```
/**

    C++ Superset -- Example #6

    Outputs:

    Test1... test2... this is a test"s" (33) out of range [0, 33[
    #0 main in memoryleak6.cpp line 32
    #1 main in memoryleak6.cpp line 29
    #2 main in memoryleak6.cpp line 26

 */



#include <string.h>

#include <iostream>



using namespace std;



int main()
{
    // Buffer overflow implicit detection:
    char * s = strdup("Test1... test2... this is a test");
```

```
    try
    {
        while (true)
        {
            cout << * s ++ << flush;
        }
    }
    catch (exception& e)
    {
        cout << e.what() << endl;
    }

    return 0;
}
```

Converting native 2D arrays into internal arrays

Because native pointers are converted by C++ Superset into a complex smart pointer type, arguments of functions using such array of native pointers will need to be converted iteratively into that array of smart pointer type before being further accessed.

This is true for the "*main()*" function and other functions declared as "*extern C++*" as we will see in the next section.

The following example was compiled with the following command:

```
fcxxss.sh -DBOOST_DISABLE_THREADS memoryleak7.cpp
```

```
/**

    C++ Superset -- Example #7

    Outputs:
```

```
    Arg #0 = ./memoryleak7
    Arg #1 = test1
    Arg #2 = test2
    Arg #3 = test3

 */


#include <iostream>



using namespace std;



int main(int argc, char * argv_[])
{
    // Transfering 2D external buffers into internal buffers:
    char ** argv = new char *[argc];

    for (int i = 0; i < argc; ++ i)
        argv[i] = argv_[i];

    for (int i = 0; i < argc; ++ i)
        cout << "Arg #" << i << " = " << argv[i] << endl;

    return 0;
}
```

Keeping native argument and return types with “extern”

Defining a function with the "*extern*" keyword tells C++ Superset not to alter argument pointer type, return pointer type and to keep them native.

The following example was compiled with the following command:

```
fcxxss.sh -DBOOST_DISABLE_THREADS memoryleak8.cpp
```

```
/**

    C++ Superset -- Example #8

    Outputs:

    Arg #0 = ./memoryleak8
    Arg #1 = test1
    Arg #2 = test2

*/



#include <iostream>



using namespace std;



// "extern" will not alter parameter types:
extern “C++” void foo(int argc, char * argv_[])
{
    for (int i = 0; i < argc; ++ i)
        cout << "Arg #" << i << " = " << argv_[i] << endl;
}
```

```
int main(int argc, char * argv_[])

{

    // Transfering 2D external buffers into the exact same argument types:

    foo(argc, argv_);


    return 0;

}
```

User headers vs. system headers

Headers located in the project's folder will be altered in order to be consistent with the source files. On the other hand, headers located in any of the system folders will not be affected.

You can also define such extra system folder to the compiler using any of the "*-isystem*" argument derivative. This way the aforementioned folder will keep the code of its headers native.

The following example was compiled with the following command:

```
fcxxss.sh -DBOOST_DISABLE_THREADS memoryleak9.cpp
```

```
/**

    C++ Superset -- Example #9

    Outputs:

    30

*/



#include <iostream>
```

```
#include "memoryleak9.hpp"



using namespace std;



int main()

{

    F * p = new F;


    delete p; // no-op


    cout << p->j.i << endl;


    return 0;

}
```

User header in question:

```
/**

    C++ Superset -- Example #9

    Outputs:

    30

*/



#ifndef MEMORYLEAK9_HPP

#define MEMORYLEAK9_HPP
```

```
struct A
{
    static int const q = 10;

    int i;
    int j;
};

struct B : virtual A
{
    int k;
    int l;
};

struct C : virtual A
{
    int m;
    int n;
};

struct D : B
{
    int o;
    int p;
};

struct E : C
{
    int o;
    int p;
```

```
};


struct F : D, E
{
    int r = 20;


    struct nested
    {
        int i;


        nested(int a) : i(a) {}
    } j = q + r;
};




#endif
```

Thread safety

It is shown in the following example that C++ Superset generates thread safe code.

The following example was compiled with the following command:

```
fcxxss.sh memoryleak10.cpp -lpthread
```

```
/**

    C++ Superset -- Example #10

    Outputs:

    foo
```

```
    bar
    [...]
    Exiting.
    Exiting: foo
    Exiting: bar

 */



#include <string.h>


#include <iostream>
#include <thread>



using namespace std;



static char const * p = strdup(__FILE__);


extern “C++” void foo()
{
    for (int i = 0; i < 1000; ++ i)
    {
        p = strdup(“foo”);
    }


    cout << "Exiting: " << “foo” << endl;
}


extern “C++” void bar()
{
```

```
    for (int i = 0; i < 1000; ++ i)
    {
        p = strdup(“bar”);
    }

    cout << "Exiting: " << “bar” << endl;
}

int main()
{
    // Thread safety demonstration:
    thread t1(foo);
    thread t2(bar);

    for (int i = 0; i < 1000; ++ i)
        cout << p << endl;

    cout << "Exiting." << endl;

    t1.join();
    t2.join();

    return 0;
}
```

Reports

Compilation

To generate a report from which the following statistics are reported:

- Memory leak;
- Double free;
- Use after free;
- Out of bounds;

The code will have to be recompiled using an extra flag:

```
fcxxss.sh -DBOOST_REPORT -DBOOST_DISABLE_THREADS memoryleak2.cpp
```

Generation

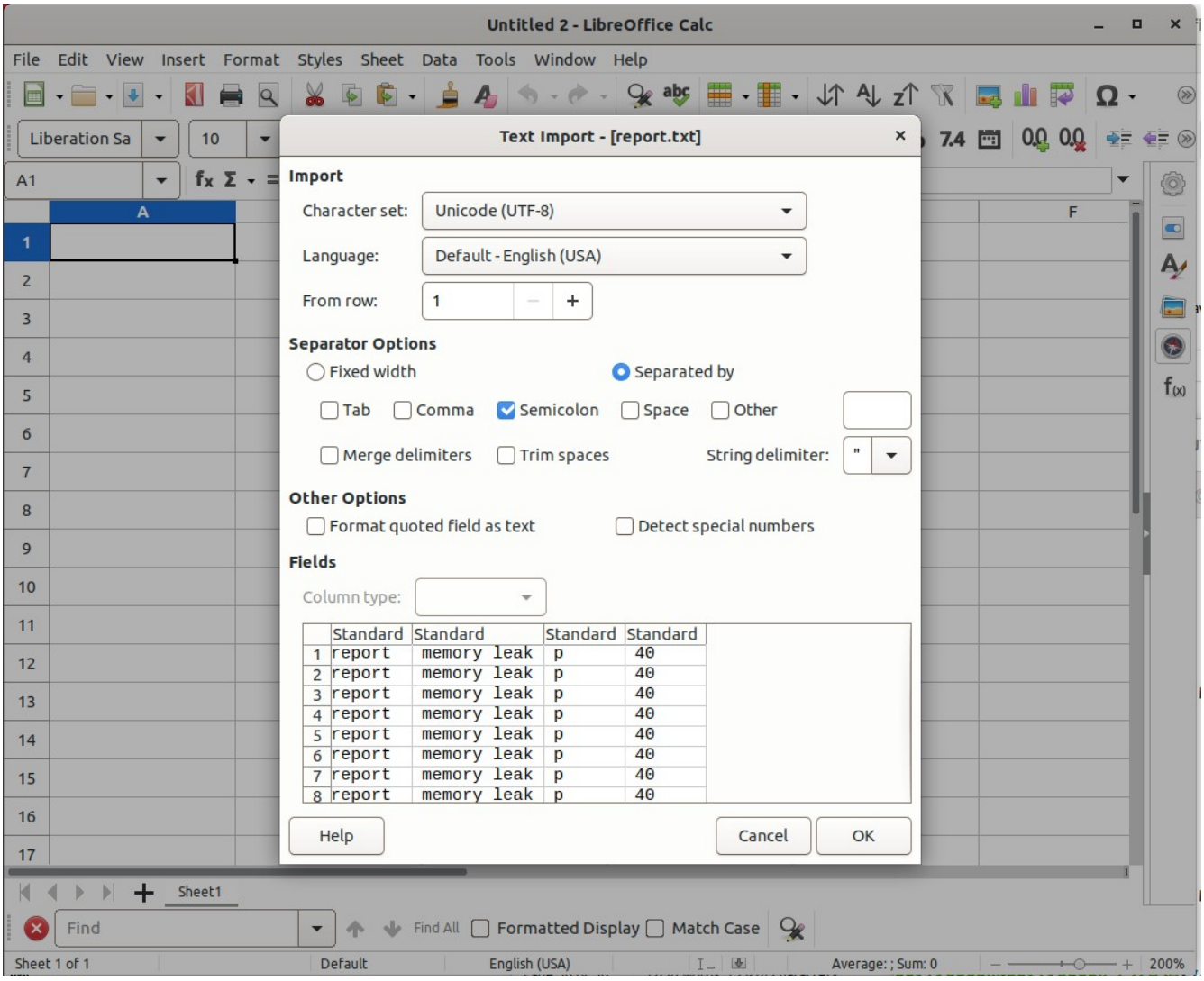
Now the statistics will be streamed out on the error stream that will have to be redirected into a local file:

```
./memoryleak2 2> report.txt
```

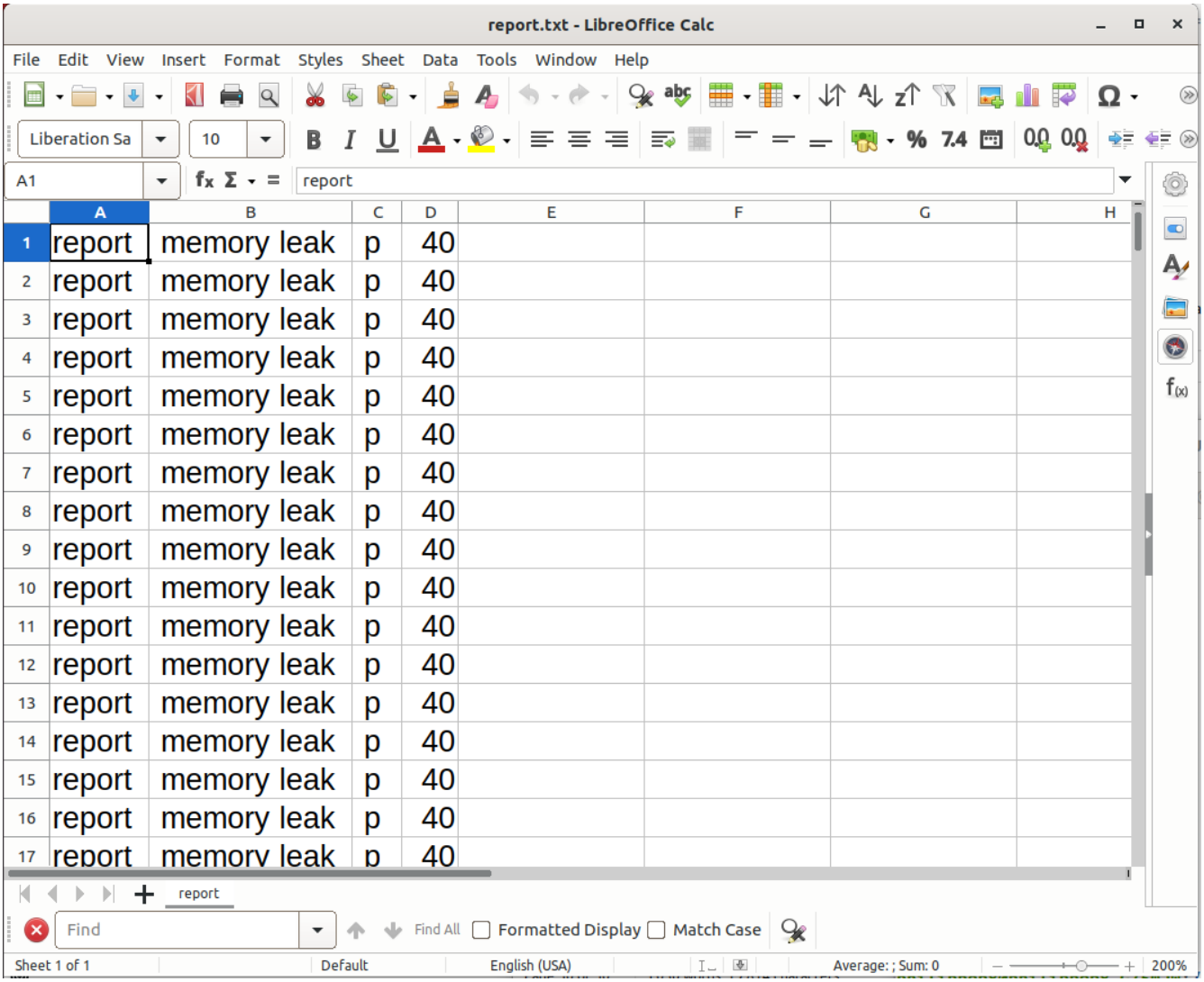
Presentation

Using your favorite spreasheet editor, we can easily compile the statistics and generate the respective graphic to represent the data.

For example using LibreOffice, you import the data with the following settings:



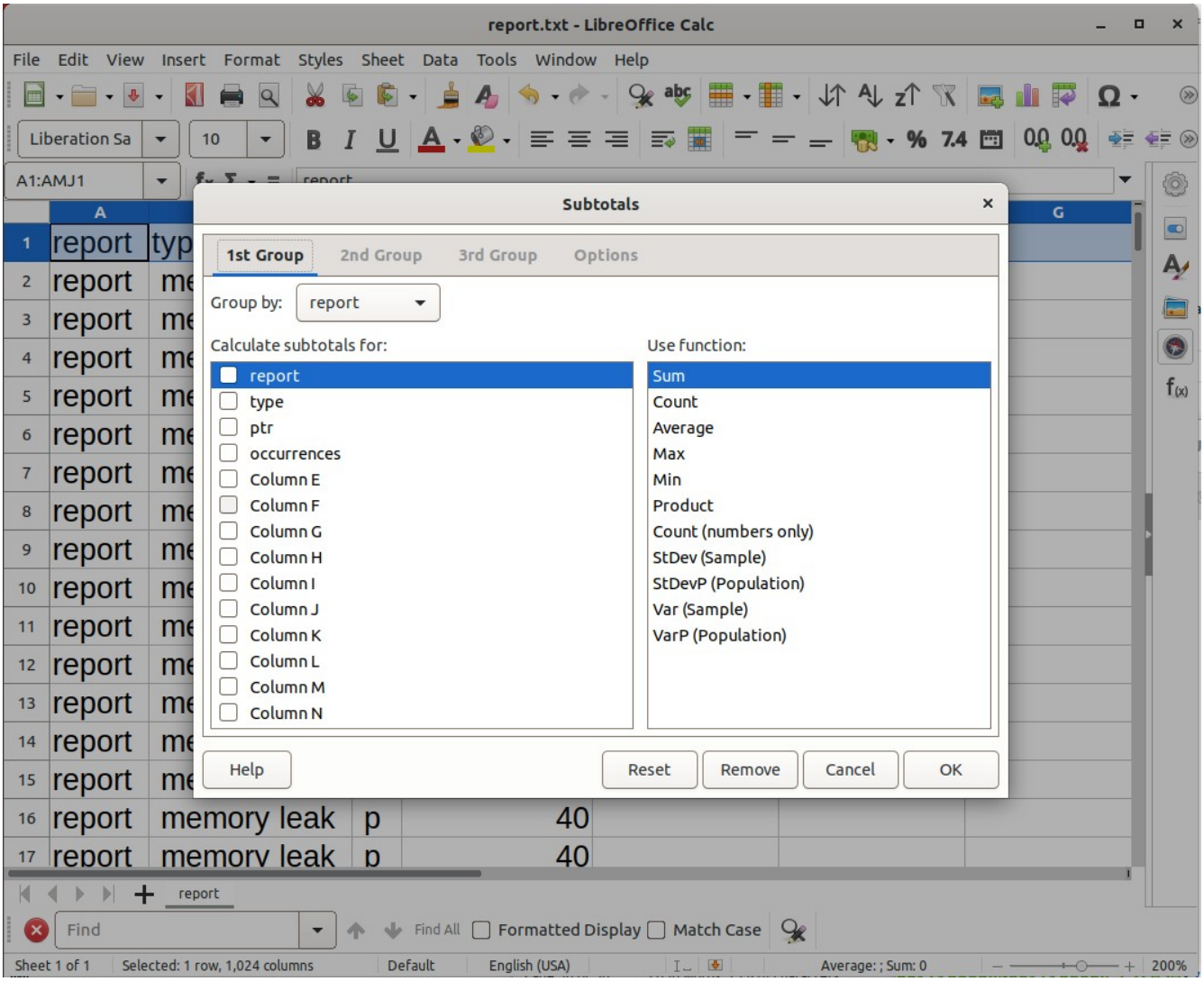
Which will result with:



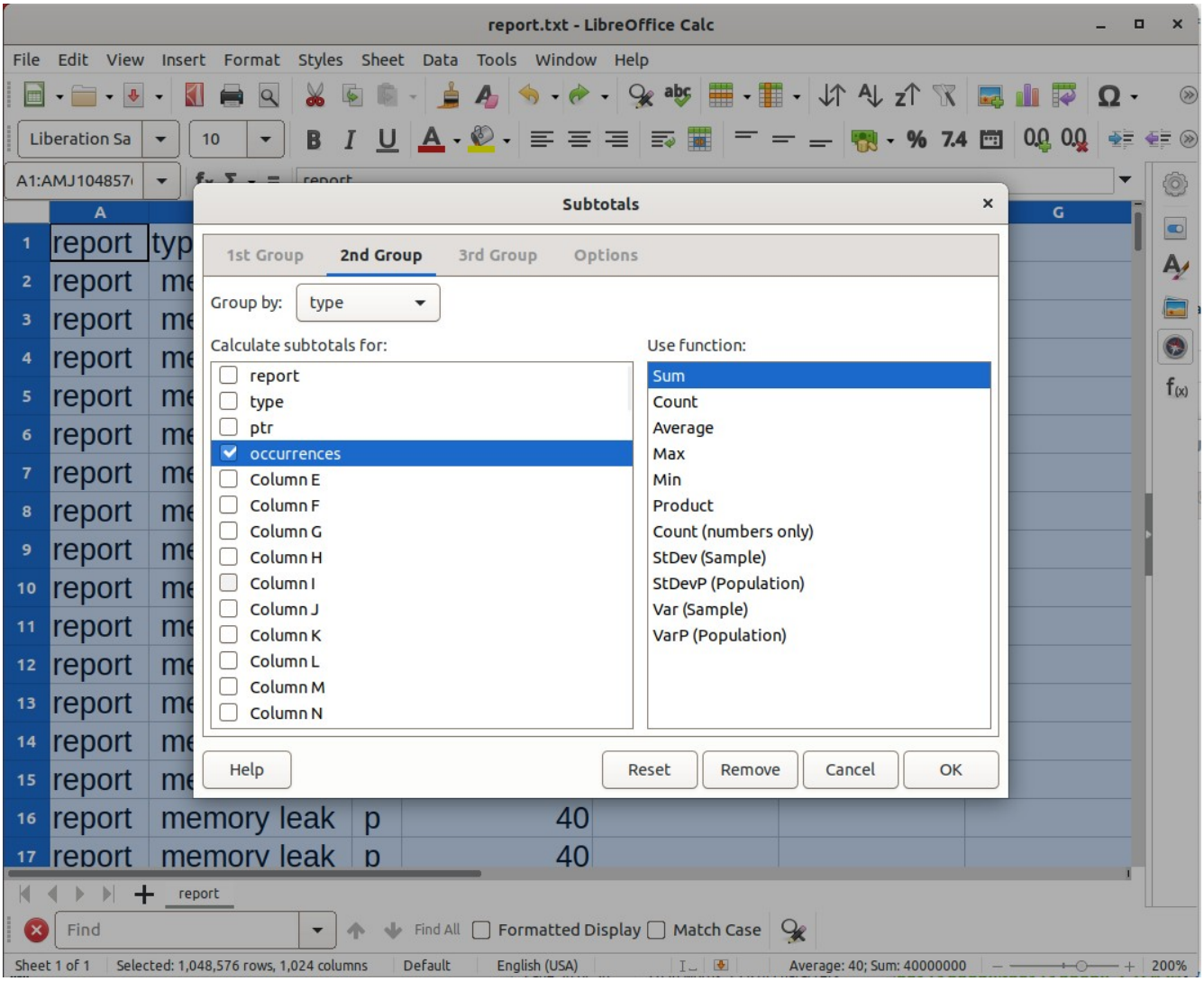
Now you'll have to add the title at the top row with the following text (without quotes):

"report, type, ptr, occurrences"

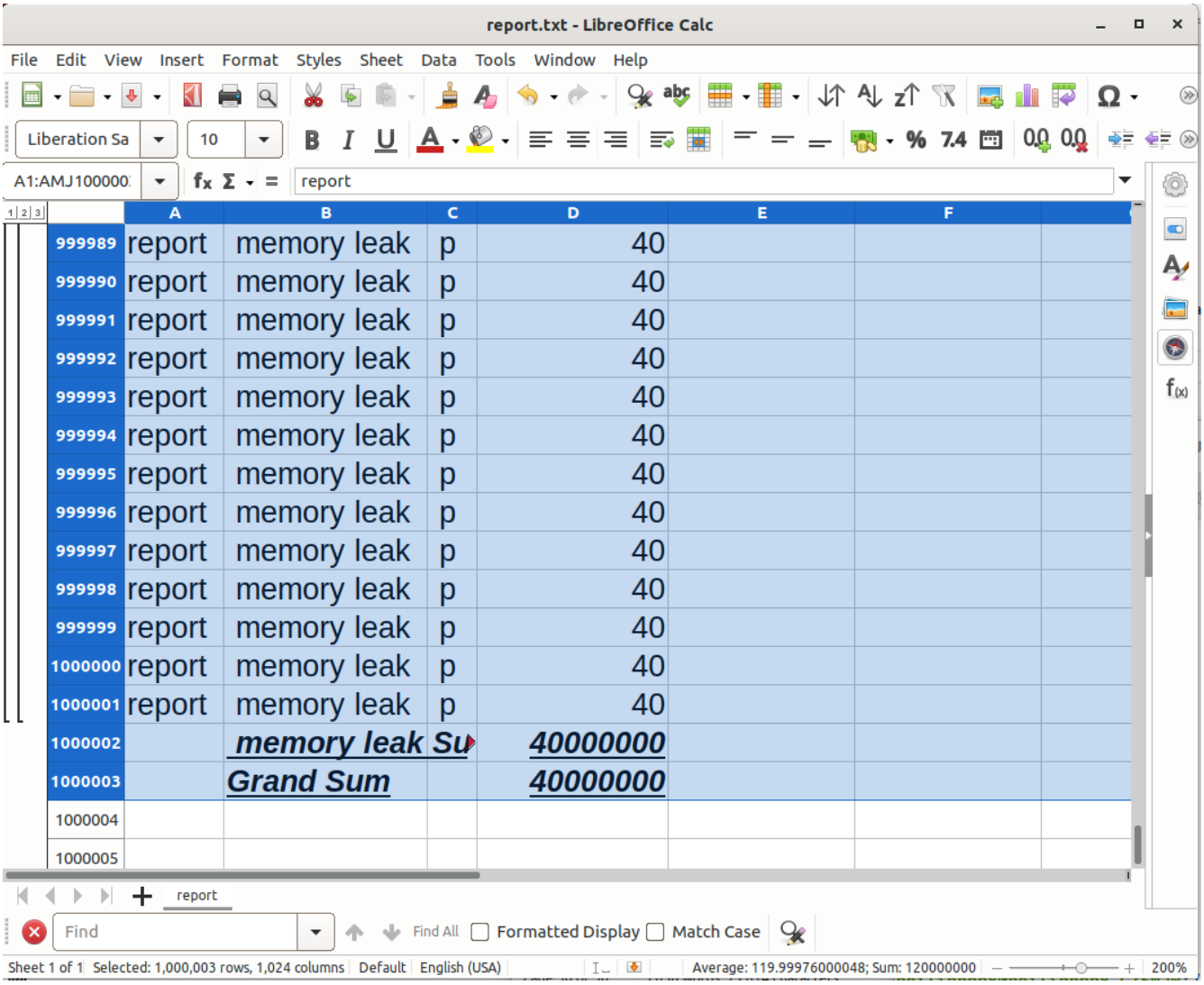
You then select "Data → Subtotals...":



You then select the 2nd group, followed by “Group by” → “report” and finally you select the “occurrences” in the “Calculate subtotals for:” column:



Which will result with the general report:



This will give you the total (in “bytes” or “occurrences”) of all the fixes C++ Superset performs implicitly.