
Fornux C++ Superset

User Manual

Deterministic memory management for C++, without a garbage collector

Version 6.4 • September 2026

Services Informatiques Fornux

20 Poirier St., Gatineau, Quebec, Canada, J8V 1A6

<https://www.fornux.com/>

Patent US11288049B2: *Source to source compiler, compilation method, and computer-readable medium for predictable memory management.*

Contents

1	Introduction	1
1.1	Why another memory manager?	1
1.2	How it works	1
2	Installation	3
2.1	Packages	3
2.2	Licence file	3
2.3	Environment	3
2.4	Checking the installation	4
3	Getting started	5
3.1	A first program	5
3.2	Objects that outlive their function	6
3.3	What the transformer produces	6
4	Licences	8
4.1	Licence types	8
4.2	One file at a time	8
4.3	Server licences and the machine ID	8
4.4	Messages	9
5	Building projects	10
5.1	Make and CMake	10
5.2	The build directory	11
5.3	Language standard	11
6	The memory model	12
6.1	Root pointers, nodes and proxies	12
6.2	When memory is released	12
6.3	Out-of-bounds and null accesses	12
6.4	Exceptions	13

6.5	Threads	13
7	Controlling the conversion	14
7.1	Making the attributes available	14
7.2	The attributes	15
7.3	Defaults	15
7.4	Example	16
8	Working with C and system libraries	18
8.1	What is automatic	18
8.2	The <code>&*</code> idiom	18
8.3	Buffers that C writes pointers into	18
8.4	C functions that write through <code>T **</code>	19
8.5	Ownership handed to the standard library	20
9	Choosing the smart pointer	21
10	Allocation and performance	22
10.1	Allocators	22
10.2	Build options	22
10.3	Measured performance	23
10.4	Which settings to use	23
10.5	Writing your own <code>operator new</code>	23
11	Reference	24
11.1	Driver	24
11.2	Licence	24
11.3	Environment variables	24
11.4	Library macros	24
12	Troubleshooting	26
12.1	Known limitations	26
13	Legal notices	27

Chapter 1

Introduction

The Fornux C++ Superset is a *source-to-source compiler*. You write ordinary C++; it rewrites your sources so that raw pointers and arrays become smart pointers backed by a deterministic memory manager, then compiles the result with clang. There is no garbage collector, no pause and no runtime to link: memory is released at predictable points, when the scope that owns it ends.

What you get, without changing your code:

- **No leaks, including cycles.** Objects that are no longer reachable are released when the scope that created them ends — also when they point at each other.
- **No dangling pointers.** An object stays alive while anything can still reach it, including after the function that created it has returned.
- **No crashes on bad accesses.** Reading past the end of an array or through a null pointer yields a zeroed default value instead of undefined behaviour.

1.1 Why another memory manager?

Memory managers have so far come in two families, and neither covers every program.

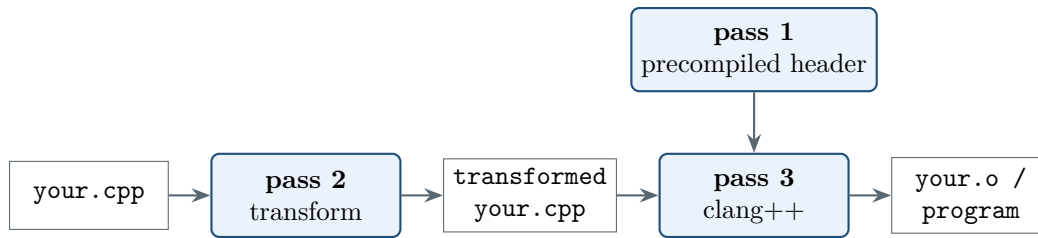
Garbage collection (Java, C#, the Boehm collector for C and C++) finds unreferenced blocks and frees them *later*. It handles cycles, but destruction is not deterministic: destructors run at an unpredictable time, if at all, and collection pauses are visible — on a user interface, for example.

Reference counting (`std::shared_ptr`) frees an object the moment its last reference goes away, so it is deterministic — but two objects that refer to each other keep each other alive forever: a cycle is never released.

The root pointer used by the Superset combines both strengths. Every object is allocated in a *node* that belongs to the scope that created it. When that scope ends, its nodes are examined: those still reachable from outside are handed to the enclosing scope, and all the others — cycles included — are destroyed immediately, in a deterministic order.

1.2 How it works

The driver, `fcxxss.sh`, stands in for your C++ compiler. Each source file goes through three passes:

**Pass 1**

builds a precompiled header of the runtime library (`fcxxss.hpp`, which includes `boost/smart_ptr/root_ptr.hpp`). It is built once per build directory and set of compiler flags, then reused.

Pass 2

parses your source with the Clang front end and writes a transformed copy into the build directory (`.build-fcxxss` by default).

Pass 3

compiles the transformed copy with clang against the precompiled header. Diagnostics point at your *original* file and line.

Only your own code is transformed. System and third-party headers — the C and C++ standard libraries, anything under `/usr/` — are used as they are, and the transformed code talks to them through the bridges described in Chapter 8.

Chapter 2

Installation

2.1 Packages

The Superset is distributed as two Debian packages:

Package	Architecture	Size	Contents
<code>fcxxss</code>	amd64, arm64	16-35 MB	the transformer and the <code>fcxxss.sh</code> driver
<code>fcxxss-headers</code>	all	11 MB	the runtime headers, this manual and examples

Install both with `apt`, which also pulls in `clang` (the driver compiles with the system's `clang++`):

```
sudo apt install ./fcxxss_6.4-1_amd64.deb ./fcxxss-headers_6.4-1_all.deb
```

Everything is installed under `/opt/fornux/superset`:

<code>bin/fcxxss.sh</code>	the driver you call instead of <code>clang++</code>
<code>usr/bin/fcxxss</code>	the transformer (called by the driver)
<code>usr/include/</code>	<code>fcxxss.hpp</code> , <code>boost/smart_ptr/root_ptr.hpp</code> and the rest of the runtime
<code>usr/bin/fcxxss.lic</code>	your licence file (not in the packages)
<code>doc/</code>	this manual

The package also installs man pages: `man fcxxss.sh` for the driver and `man 7 fcxxss` for a summary of this manual.

2.2 Licence file

The transformer needs your licence file. Install it next to the transformer, as `/opt/fornux/superset/usr/bin/fcxxss.lic`:

```
sudo cp fcxxss.lic /opt/fornux/superset/usr/bin/fcxxss.lic
```

Without it, every compilation stops with `License is missing`. The licence types, and how to obtain a server licence for one machine, are described in Chapter 4.

2.3 Environment

Put the driver on your `PATH`:

```
export PATH=/opt/fornux/superset/bin:$PATH
```

2.4 Checking the installation

Asked for help, the driver prints its version and the environment variables it reads. The first line, a complaint about the option itself, is expected:

```
$ fcxxss.sh --help
/opt/fornux/superset/bin/fcxxss.sh: unrecognized option '--help'
Usage: /opt/fornux/superset/bin/fcxxss.sh [options] file

Fornux C++ Superset - Source-to-Source Compiler 6.4 (for clang)

--nodeid                print this machine's ID, for a server licence

Environment:
FCXXSS_BUILD_DIR=dir    intermediate files (default: ./build-fcxxss)
FCXXSS_NO_THREADS=1     single-threaded program: build with BOOST_DISABLE_THREADS
FCXXSS_SHARED_PTR=1     converted pointers are boost::shared_node_ptr (a std::
                        shared_ptr)
FCXXSS_POINTER_TEMPLATE=X converted pointers are X<T> (default ::boost::root_ptr)
FCXXSS_EXCEPTIONS=1     keep exceptions (default: BOOST_NO_EXCEPTIONS)
FCXXSS_NO_STATIC_ASSERT= keep static_assert (default: defined away)
FCXXSS_STDCXX=NN        C++ standard: 17 (default), 20, 23, ...
```

Chapter 3

Getting started

3.1 A first program

This program creates a million pairs of objects that point at each other and never deletes them. In standard C++ every pair leaks.

```
// A cycle that is never deleted: native C++ leaks it, the superset reclaims it.
#include <cstdio>
#include <fstream>
#include <unistd.h>

struct Node
{
    int value;
    Node * next;
    Node(int v) : value(v), next(nullptr) {}
};

static long resident_kb()
{
    long pages = 0, resident = 0;
    std::ifstream("/proc/self/statm") >> pages >> resident;
    return resident * (sysconf(_SC_PAGESIZE) / 1024);
}

static void make_cycle(int i)
{
    Node * a = new Node(i);
    Node * b = new Node(i + 1);
    a->next = b;
    b->next = a;           // a <-> b: unreachable once we return
}

int main()
{
    long before = resident_kb();
    for (int i = 0; i < 1000000; ++i)
        make_cycle(i);
    std::printf("memory grown: %ld kB\n", resident_kb() - before);
}
```

Compile it with `fcxxss.sh` exactly as you would with `clang++`:

```
$ fcxxss.sh -O2 cycles.cpp -o cycles
cycles.cpp: pass 1
cycles.cpp: pass 2
```

```
cycles.cpp: pass 3
$ ./cycles
memory grown: 0 kB
```

The same source compiled with plain `clang++` reports `memory grown: 63824 kB`. Each pair is released when `make_cycle` returns, because nothing outside the function can reach it any more.

3.2 Objects that outlive their function

Releasing at scope exit would be useless if it released objects that are still in use. An object returned by a function, or stored somewhere longer-lived, is handed to the caller:

```
// An object returned from a function outlives the function's scope.
#include <cstdio>

struct Node
{
    int value;
    Node * next;
    Node(int v, Node * n) : value(v), next(n) {}
};

Node * push(Node * head, int v)
{
    return new Node(v, head);
}

int main()
{
    Node * list = nullptr;
    for (int i = 1; i <= 3; ++i)
        list = push(list, i * 10);
    for (Node * n = list; n; n = n->next)
        std::printf("%d ", n->value);
    std::printf("\n");
}
```

```
$ fcxxss.sh -O2 escape.cpp -o escape && ./escape
30 20 10
```

3.3 What the transformer produces

You never need to read the transformed source, but it helps to know what is in it. This is `push` and `main` from the program above, as written to `.build-fcxxss/.../escape.cpp` (#line directives removed):

```
::boost::root_ptr<Node> push(::boost::node_proxy const & __y,
                             ::boost::root_ptr<Node> head, int v)
{
```

```

    {
        ::boost::node_proxy const & __x = __y();
        ::boost::node_proxy const __y(__FILE__, __func__, __LINE__, & __x());
        return ::boost::root_ptr<Node>{__y(), new (__y()) ::boost::node<Node>{__y(), v,
        head}};
    }
}

int main()
{
    {
        ::boost::node_proxy const & __x = __y();
        ::boost::node_proxy const __y(__FILE__, __func__, __LINE__, & __x());

        ::boost::root_ptr<Node> list = ::boost::root_ptr<Node>{__y(), ::boost::root_ptr
<::std::nullptr_t>(__y(), nullptr)};
        for (int i = 1; i <= 3; ++i)
            list = push(__y(), list, i * 10);
        ...
    }
}

```

Three things changed:

- every `Node *` became `::boost::root_ptr<Node>`;
- `new Node(...)` allocates a `::boost::node<Node>`;
- each function received a `node_proxy` parameter, `__y`, and opens a proxy of its own chained to the caller's. That proxy is the scope that owns the function's allocations.

The transformed file also carries a comment before each function listing the attributes the transformer applied to it implicitly, for example:

```
/* fcxxss: FCXXSS_NO_PROXY (main), FCXXSS_CONVERTED_BODY */
```

The attributes themselves are described in Chapter 7.

Chapter 4

Licences

Every run of the transformer — once per source file — reads the licence file `/opt/fornux/superset/usr/bin/fcxxxs.lic` and checks it before doing anything else.

4.1 Licence types

Type	Machines	Duration	Parallel builds
trial	any	limited (typically 31 days)	no: one file at a time
single	any	unlimited	no: one file at a time
server	the one machine it was issued to	unlimited	yes

Trial

Each run reports the days left, for example `Enabling the 'fcxxxs' functionality for 31 more day(s).`, and stops with `License is expired.` afterwards.

Single

Each run prints `This software is to be used under GPL regulations.` and pauses for one second.

Server

Locked to the machine ID it was issued for; no limit on parallel runs.

4.2 One file at a time

With a trial or single licence, only one transformation may run at a time on a machine. A second one started meanwhile stops with `Concurrent processes`. Build these projects *sequentially*:

```
make                # not: make -j
cmake --build . -j 1 # with CMake
```

A server licence has no such limit: `make -j` works as usual.

4.3 Server licences and the machine ID

A server licence is issued to one machine's ID. Print it with `fcxxxs.sh --nodeid` on the machine that will run the builds, and send it to Fornux:

```
$ fcxxxs.sh --nodeid
2C4A7EB0D22C3B80A532862C1D0CBF8090D6E3308B683F589CAED8D03128D8C0
```

The ID is derived from the host name and the network interfaces (and, on Intel, the processor), so it changes if the machine is renamed or its network hardware is replaced; ask for a new licence then. An ID taken on one architecture does not match the other: run `fcxxss.sh --nodeid` on the machine itself. It needs no licence.

4.4 Messages

Message	Meaning
License is missing.	no <code>fcxxss.lic</code> next to the transformer (<code>/opt/fornux/superset/usr/bin/</code>)
License is not valid.	the file is damaged or was not issued by Fornux
License is expired.	the trial period is over
License is for the wrong node.	a server licence on a machine whose ID differs from the one it was issued to
License is missing entitlement.	the licence names no known type
Concurrent processes.	a trial or single licence, and another transformation is running: build with <code>make</code> , not <code>make -j</code>

Building projects

5.1 Make and CMake

`fcxxss.sh` accepts the usual compiler and linker options (`-c`, `-o`, `-I`, `-D`, `-O`, `-g`, `-l`, `-L`, `-std`, ...) and passes them on to clang. Use it as the C++ compiler of your build system.

With Make:

```
make CXX=/opt/fornux/superset/bin/fcxxss.sh
```

With a trial or single licence, build without `-j`; a server licence allows parallel builds (Chapter 4).

With CMake, choose it when you configure:

```
cmake -DCMAKE_CXX_COMPILER=/opt/fornux/superset/bin/fcxxss.sh ..
make
```

CMake's own compiler checks are recognised and passed straight to `clang++`, so the configure step behaves as it would with plain clang. A two-file project built this way:

```
$ cmake -DCMAKE_CXX_COMPILER=/opt/fornux/superset/bin/fcxxss.sh .. && make
[ 33%] Building CXX object CMakeFiles/shapes.dir/main.cpp.o
.../main.cpp: pass 1
.../main.cpp: pass 2
.../main.cpp: pass 3
[ 66%] Building CXX object CMakeFiles/shapes.dir/shape.cpp.o
.../shape.cpp: pass 2
.../shape.cpp: pass 3
[100%] Linking CXX executable shapes
$ ./shapes
total area 26
```

Pass 1 ran only once: the second file reused the precompiled header.

Every file of a program goes through the Superset

A transformed function takes a `node_proxy` parameter and transformed classes hold `root_ptr` members, so transformed and untransformed C++ code cannot call each other directly. Build all the C++ sources of a program with `fcxxss.sh`. Precompiled libraries — the C library, the C++ standard library, system packages — are fine: they are reached through their headers, which are not transformed.

5.2 The build directory

Intermediate files go into `.build-fcxxxss` in the *current working directory* — the build directory of an out-of-tree build — so the source tree stays clean. Set `FCXXXSS_BUILD_DIR` to use another one:

```
FCXXXSS_BUILD_DIR=/tmp/mybuild fcxxxss.sh -c foo.cpp
```

Inside it, each transformed source keeps its name under the absolute path it came from, so two projects compiling a file of the same name cannot overwrite each other. For example, `/home/me/proj/foo.cpp` is transformed into `.build-fcxxxss/home/me/proj/foo.cpp`.

The precompiled header is shared by every file compiled with the same flags in that directory; changing a `-D`, `-I` or `-std` produces a new one. Parallel builds (`make -j`) are safe: building the header is locked. To force it to be rebuilt, pass `--clean` or delete the build directory.

5.3 Language standard

Sources are compiled as C++17. Set `FCXXXSS_STDCXX` to the number of another standard — `FCXXXSS_STDCXX=20` for C++20, `FCXXXSS_STDCXX=23` for C++23. A `-std` option on the command line is ignored, so this is the only way to choose the standard.

The memory model

6.1 Root pointers, nodes and proxies

Three types from `boost/smart_ptr/root_ptr.hpp` carry the whole model:

`boost::node<T>`

the block that holds one object (or one array) together with its reference count. Every `new` in your code allocates one.

`boost::root_ptr<T>`

what a `T *` becomes. It points into a node and keeps it alive; it converts to `T *`, supports `*`, `->`, `[]`, comparisons and pointer arithmetic, and is `nullptr`-compatible. Arrays `T a[N]` become `boost::root_array<T, N>`, a `root_ptr` over a node holding a `std::array<T, N>`.

`boost::node_proxy`

a scope. Each function body opens one, chained to its caller's; every node allocated in the function belongs to it.

6.2 When memory is released

1. An object whose reference count drops to zero is released at once, as with `std::shared_ptr`.
2. When a scope ends, its proxy examines the nodes it owns. Those still reachable from outside — returned, stored in a longer-lived object, referenced from a global — are handed to the enclosing scope. Every other node is destroyed immediately, *including cycles*.
3. Nodes handed out of `main` belong to the global scope, which releases them when the program ends.

Destructors run as part of these steps, so their timing is deterministic and the same on every run. An exception thrown by a destructor during release is rethrown when the scope ends, unless the stack is already unwinding.

6.3 Out-of-bounds and null accesses

A legacy program that crashes is a program that stops serving its users. The Superset therefore turns the most common memory errors into well-defined behaviour:

```
// Out-of-bounds and null accesses: undefined behaviour natively, a default  
// value in the superset.  
#include <cstdio>
```

```

struct Point { int x, y; };

int main()
{
    int a[4] = { 1, 2, 3, 4 };
    int i = 7;                // past the end
    std::printf("a[7] = %d\n", a[i]);

    Point * p = nullptr;
    std::printf("p->x = %d\n", p->x);
    std::printf("still running\n");
}

```

```

$ clang++ -O0 bounds.cpp -o bounds && ./bounds
Segmentation fault (core dumped)
$ fcxxss.sh -O2 bounds.cpp -o bounds && ./bounds
a[7] = 0
p->x = 0
still running

```

Indexing a converted array past its end, or dereferencing a null `root_ptr`, returns a reference to a zero-initialised *default instance* of the type — one shared, statically allocated object per type, never constructed. Reading it yields zeros; writing to it is harmless. The checks are always on.

6.4 Exceptions

By default the runtime is built with `BOOST_NO_EXCEPTIONS`: it never throws, which also lets it be used in code compiled with `-fno-exceptions`. Your own exceptions are not affected. Set `FCXXSS_EXCEPTIONS=1` to build the runtime with exceptions enabled.

6.5 Threads

Proxies are per thread: each thread's functions form their own chain of scopes. Objects may be shared between threads like any pointer.

By default every `root_ptr` operation takes one global lock, which is safe in every program but serialises threads that allocate heavily. Two options change this:

- A **single-threaded** program can drop all locking with `FCXXSS_NO_THREADS=1`: `root_ptr` operations become several times faster. The program must then not use threads at all.
- A program with many allocating threads can define `BOOST_ROOT_PTR_STRIPED_LOCKS`, which replaces the global lock with 1024 address-striped locks, together with `BOOST_PAGE_ALLOCATOR_THREAD_CACHE`. See Chapter 10 for the trade-off.

Controlling the conversion

The defaults convert everything that can safely be converted. Attributes let you override them per function, variable, class or namespace.

7.1 Making the attributes available

The attributes are macros that expand to clang annotations. Add this header to your project (it is also in `doc/examples/`) and include it where you use them. Under another compiler the macros expand to nothing, so the same sources still build normally.

```
// Fornux C++ Superset attributes, for a project's own sources.
// The transformer reads them as clang annotations; a plain compiler ignores them.
#ifndef FCXXSS_ATTRIBUTES_H
#define FCXXSS_ATTRIBUTES_H

#if defined(__clang__)
#define FCXXSS_ANNOTATE(x) __attribute__((annotate(x)))
#else
#define FCXXSS_ANNOTATE(x)
#endif

#ifndef FCXXSS_RAW
#define FCXXSS_RAW FCXXSS_ANNOTATE("fcxxss::raw")
#endif
#ifndef FCXXSS_PROXY
#define FCXXSS_PROXY FCXXSS_ANNOTATE("fcxxss::proxy")
#endif
#ifndef FCXXSS_NO_PROXY
#define FCXXSS_NO_PROXY FCXXSS_ANNOTATE("fcxxss::no_proxy")
#endif
#ifndef FCXXSS_RAW_PARAMS
#define FCXXSS_RAW_PARAMS FCXXSS_ANNOTATE("fcxxss::raw_params")
#endif
#ifndef FCXXSS_CONVERTED_PARAMS
#define FCXXSS_CONVERTED_PARAMS FCXXSS_ANNOTATE("fcxxss::converted_params")
#endif
#ifndef FCXXSS_RAW_BODY
#define FCXXSS_RAW_BODY FCXXSS_ANNOTATE("fcxxss::raw_body")
#endif
#ifndef FCXXSS_CONVERTED_BODY
#define FCXXSS_CONVERTED_BODY FCXXSS_ANNOTATE("fcxxss::converted_body")
#endif
#ifndef FCXXSS_POINTER
#define FCXXSS_POINTER(X) FCXXSS_ANNOTATE("fcxxss::pointer=" #X)
#endif
```

```

#ifndef FCXXSS_SHARED_PTR
#define FCXXSS_SHARED_PTR FCXXSS_POINTER(::boost::shared_node_ptr)
#endif
#ifndef FCXXSS_ROOT_PTR
#define FCXXSS_ROOT_PTR FCXXSS_POINTER(::boost::root_ptr)
#endif

#endif

```

7.2 The attributes

Attribute	Effect
FCXXSS_RAW	The declaration keeps its written form: no proxy parameter, no converted pointers. Use it on a variable that C code writes into, on a typedef (<code>typedef void * RawSlot FCXXSS_RAW;</code> — after the name), on a function whose signature is fixed from outside, or on a whole namespace.
FCXXSS_PROXY	The function takes a proxy parameter and opens a scope of its own. Operators and <code>constexpr</code> functions have none by default.
FCXXSS_NO_PROXY	The function takes no proxy parameter; its body uses the global one.
FCXXSS_RAW_PARAMS	Pointer parameters keep their written types; callers holding a <code>root_ptr</code> pass the raw pointer.
FCXXSS_CONVERTED_PARAMS	Pointer parameters are converted where the default keeps them as written.
FCXXSS_RAW_BODY	Nothing in the function body is converted: locals, literals and <code>new</code> expressions stay raw.
FCXXSS_CONVERTED_BODY	The body is converted where the default keeps it raw.
FCXXSS_POINTER(X)	Converted pointers are spelled <code>X<T></code> instead of <code>boost::root_ptr<T></code> . On a function: its parameters, return type and body; on a class: its members; on a namespace: everything in it.
FCXXSS_SHARED_PTR	Short for <code>FCXXSS_POINTER(::boost::shared_node_ptr)</code> : a reference-counted <code>std::shared_ptr</code> . See Chapter 9.
FCXXSS_ROOT_PTR	Back to <code>boost::root_ptr</code> inside a shared class or namespace.

7.3 Defaults

You need attributes only where the defaults are wrong for your code. The transformer applies these on its own and records them in the comment it writes before each function:

Declaration	Default
ordinary functions and constructors	proxy parameter, converted parameters and body
<code>main</code>	no proxy parameter (it opens the first scope)
operators	no proxy parameter; operator syntax kept (<code>a + b</code>)
<code>constexpr</code> functions	no proxy parameter, so they stay usable in constant expressions
<code>extern "C"</code> functions	no proxy parameter (C linkage)
functions passed to C as callbacks	parameters kept raw
replaceable global <code>operator new/delete</code>	unchanged
class and placement <code>operator new</code>	take the proxy as their second parameter
system headers	never transformed
<code>std::atomic<T *></code>	kept raw (an atomic needs a trivially copyable type)

7.4 Example

```

// Per-declaration control of the conversion.
#include <cstdio>
#include "fcxxss_attributes.h"

struct Vec
{
    double x, y;
};

// Operators take no node_proxy by default and keep operator syntax.
Vec operator + (Vec a, Vec b) { return Vec{ a.x + b.x, a.y + b.y }; }

// Reference counted instead of root_ptr: a std::shared_ptr underneath.
struct FCXXSS_SHARED_PTR Widget
{
    int id;
    Widget * self() { return this; } // shares ownership (enable_shared_from_this)
};

// A function whose signature must stay exactly as written.
FCXXSS_RAW int length(const char * s)
{
    int n = 0;
    while (s[n]) ++n;
    return n;
}

int main()
{
    Vec v = Vec{ 1, 2 } + Vec{ 3, 4 };
    Widget * w = new Widget{ 7 };
    Widget * same = w->self();
    std::printf("%g %g, widget %d, length %d\n", v.x, v.y, same->id, length("raw"));
}

```

```
$ fcxxss.sh -O2 attributes.cpp -o attributes && ./attributes  
4 6, widget 7, length 3
```

Working with C and system libraries

System headers are not transformed, so their functions keep raw-pointer signatures. Most calls just work; this chapter covers the cases that need a word from you.

8.1 What is automatic

- A `root_ptr` converts implicitly to `T *`, so it can be passed where a C function expects a raw pointer.
- A function you hand to C as a callback (`qsort`'s comparator, a signal handler) keeps the signature C expects.
- A local whose address is passed to a C out-parameter of type `T **` (`strtol`'s `endptr`) stays a raw pointer.
- `strdup`, `strndup`, `memcpy`, `memset` and `free` are routed to versions that understand `root_ptr`.

8.2 The `&*` idiom

When a call needs the plain address and the conversion does not happen on its own — a template parameter deduces `root_ptr` instead of a pointer, or a class with its own conversion operator is involved — write `&*p`. `*p` is a `T &`, so `&*p` is a `T *`, and it costs nothing:

```
std::qsort(&*v, 5, sizeof(int), by_value);
```

8.3 Buffers that C writes pointers into

A `root_ptr` is larger than a raw pointer, so an *array of pointers* that a C function fills cannot be converted — the layouts differ. Mark such a buffer `FCXXSS_RAW`. The transformer does not guess: without the attribute the call does not compile.

```
// Calling C: callbacks, out-parameters, raw buffers and the '&*' idiom.
#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <execinfo.h>
#include "fcxxss_attributes.h"

extern "C" int by_value(const void * l, const void * r)    // a C callback
{
    return *(const int *) l - *(const int *) r;
```

```

}

int main()
{
    int v[5] = { 4, 1, 5, 2, 3 };
    std::qsort(&*v, 5, sizeof(int), by_value);           // '&*v': the raw address
    for (int x : v) std::printf("%d ", x);
    std::printf("\n");

    const char * text = "42 apples";
    char * end;                                         // its address goes to C:
    long n = std::strtol(&*text, &end, 10);            // kept raw automatically
    std::printf("%ld, rest '%s'\n", n, end);

    FCXXSS_RAW void * frames[8];                       // a buffer C fills with
    int depth = backtrace(frames, 8);                  // raw pointers
    std::printf("backtrace depth > 0: %s\n", depth > 0 ? "yes" : "no");

    char * s = strdup("superset");                     // a root_ptr-aware strdup
    std::printf("%s (%zu)\n", s, std::strlen(&*s));
}

```

```

$ fcxxss.sh -O2 interop.cpp -o interop && ./interop
1 2 3 4 5
42, rest ' apples'
backtrace depth > 0: yes
superset (8)

```

Without FCXXSS_RAW on frames:

```

interop.cpp:26:21: error: no matching function for call to 'backtrace'

```

8.4 C functions that write through T **

A function of your own that fills a buffer of raw pointers through a T ** parameter is best written in C: put it in a .c file compiled by your C compiler, and declare it FCXXSS_RAW in its header so the C++ side keeps the C signature.

```

// fill.h
#include "fcxxss_attributes.h"
#ifdef __cplusplus
extern "C" {
#endif
FCXXSS_RAW int fill(void ** slots, int n);
#ifdef __cplusplus
}
#endif

```

```

// fill.c - compiled with cc
#include "fill.h"
int fill(void ** slots, int n)

```

```
{
    for (int i = 0; i < n; ++i) slots[i] = (void *) (long) (i + 1);
    return n;
}
```

```
// main.cpp - compiled with fcxxss.sh
FCXXSS_RAW void * slots[3];
int n = fill(slots, 3);
```

```
cc -O2 -c fill.c
fcxxss.sh -O2 -c main.cpp
fcxxss.sh main.o fill.o -o program
```

8.5 Ownership handed to the standard library

A plain `new T` becomes a node, and `std::unique_ptr` or `std::shared_ptr` would later free it with a plain `delete` — corrupting the heap. The transformer refuses to compile it and says what to write instead:

```
struct Holder { std::unique_ptr<int> p; Holder(int v) : p(new int(v)) {} };
```

```
error: 'new_into_system_owner_use_make_unique<int, int &>' is unavailable: 'new T(args)
,
handed to a system owning smart pointer would be freed with a plain 'delete' on a
boost::node interior pointer. Write std::make_unique<T>(args...) instead ...
```

Write `p(std::make_unique<int>(v))` (or `std::make_shared`): the allocation then happens inside the standard library, and the smart pointer owns an ordinary block.

Choosing the smart pointer

`boost::root_ptr` is the default and the only pointer that releases cycles. `boost::shared_node_ptr` is an alternative: it *is* a `std::shared_ptr` (public base) that also accepts the forms the transformer emits, so converted code can hand it to APIs that take `std::shared_ptr`.

	<code>boost::root_ptr</code>	<code>boost::shared_node_ptr</code>
cycles	released at scope exit	never released (reference counted only)
locking	one global lock (or striped locks)	atomic counts, no global lock
4 threads allocating	slower	up to 25× faster
single-threaded	fastest	1.06–1.15× the time, 19–25% more memory
<code>std::shared_ptr</code> interop	converts both ways	is one

Choose it per class, function or namespace with `FCXXSS_SHARED_PTR`, or for a whole translation unit with `FCXXSS_SHARED_PTR=1`. A shared class automatically derives from `boost::shared_node_base`, an `enable_shared_from_this`, so a `this` pointer shares ownership correctly. `FCXXSS_ROOT_PTR` switches back inside a shared scope.

Agree across files

A class must be converted the same way in every file that uses it. The translation-unit default applies to whole files; a *class* annotation always wins over it, so annotate classes rather than relying on the environment when files are built differently.

`FCXXSS_POINTER(X)` and `FCXXSS_POINTER_TEMPLATE=X` accept any template `X` that supports `root_ptr`'s construction forms.

Allocation and performance

10.1 Allocators

Each node is allocated from a pool. The default is `boost::page_allocator_by_size`, which groups blocks of one size class in 64 KiB pages and frees in constant time. Choose another one for a whole program on the command line:

```
fcxxss.sh -DBOOST_ROOT_PTR_ALLOCATOR=std::allocator ...
```

<code>boost::page_allocator_by_size</code>	default; fastest or within 10% of the fastest in 114 of 140 measured cases
<code>boost::page_allocator_by_type</code>	one page per type: similar speed, but 2–4× slower and up to 1.9 MB more memory with many sparsely used types
<code>std::allocator</code>	glibc <code>malloc</code> ; best with many threads and no thread caches
<code>boost::fast_pool_allocator</code>	fastest create-and-drop for <code>std::unique_ptr</code> and <code>std::shared_ptr</code> in a single-threaded build
<code>boost::pool_allocator</code>	avoid : releasing many objects at once is quadratic

10.2 Build options

Option	Effect
<code>FCXXSS_NO_THREADS=1</code> (environment) or <code>-DBOOST_DISABLE_THREADS</code> <code>-DBOOST_PAGE_ALLOCATOR_THREAD_CACHE</code>	No locks, plain reference counts. Single-threaded programs only. <code>root_ptr</code> churn is 6.2× faster, cycles 2.7×. Each thread keeps a small cache of free blocks, so the page pools' lock is rarely taken. With 4 threads, allocation drops from 183 to 3.4 ns per <code>unique_ptr</code> -sized object; single-threaded churn is 1.3–3.1× faster; memory unchanged. Requires threads.
<code>-DBOOST_ROOT_PTR_STRIPED_LOCKS</code>	1024 address-striped locks instead of <code>root_ptr</code> 's global lock. With 4 threads, about 10× faster (with the thread caches); single-threaded, 1.5–2.6× slower. Requires threads.
<code>-DBOOST_ZEROIZATION</code>	Every node is cleared before its memory is released, so freed memory holds no data. No measurable cost.

All files of a program must be built with the same options.

10.3 Measured performance

Nanoseconds per operation, median of 5 runs, `page_allocator_by_size`, on an Intel Core i7-4700HQ (Linux, clang, `-O2`). Lower is better.

	<code>root_ptr</code>	<code>shared_</code> <code>node_ptr</code>	<code>unique_ptr</code>	<code>shared_ptr</code>
create and drop one object, threads	58.0	61.8	18.6	22.7
the same, <code>FCXXSS_NO_THREADS=1</code>	9.4	28.7	5.8	9.3
release 100,000 objects together	133	142	48	64
reclaim a 2-node cycle (per node)	140	—	—	—
4 threads, default build	389	239	183	167
4 threads, caches (+ striped locks for <code>root_ptr</code>)	30	12	3.4	4.3
memory for 100,000 objects (MB)	12.5	15.6	7.0	9.3

Only `root_ptr` reclaims cycles: a `std::unique_ptr` cannot form one and a `std::shared_ptr` cycle is never freed. The complete benchmark — every combination of allocator, pointer and build option — is published separately as the *Node allocator benchmark* (`ALLOCATOR_BENCHMARK.pdf`).

10.4 Which settings to use

Your program	Settings
single-threaded	<code>FCXXSS_NO_THREADS=1</code>
multi-threaded, light allocation	the defaults
many threads allocating at once	<code>-DBOOST_PAGE_ALLOCATOR_THREAD_CACHE</code> <code>-DBOOST_ROOT_PTR_STRIPED_LOCKS</code> , or <code>FCXXSS_SHARED_PTR</code> where no cycles can form
handles secrets	add <code>-DBOOST_ZEROIZATION</code>

10.5 Writing your own operator new

A class `operator new` and every placement `operator new` you declare receive the proxy as their *second* parameter, and their pointer parameters are converted like any function's. The transformer rewrites every `new` expression to pass it. For system code, which cannot pass a proxy, a plain `operator new(size_t)` that forwards to yours is generated alongside it. The replaceable global forms (`::operator new(size_t)` and its aligned and `nothrow` variants) are left unchanged.

Reference

11.1 Driver

<code>fcxxss.sh [clang options] file.cpp</code>	compile and link
<code>fcxxss.sh -c [clang options] file.cpp</code>	compile to file.o
<code>fcxxss.sh [linker options] file.o ...</code>	link

Files ending in `.c`, `.cc`, `.cxx`, `.cpp`, `.c++` or `.C` are compiled as C++; a command line without a source file is a link. Without `-o`, the output is named after the source. `--clean` rebuilds the precompiled header.

11.2 Licence

<code>/opt/fornux/superset/usr/bin/fcxxss.lic</code>	the licence file, read by every transformation
<code>fcxxss.sh --nodeid</code>	prints this machine's ID, for a server licence
<code>/tmp/fcxxss.lock</code>	held by the running transformation under a trial or single licence

11.3 Environment variables

Variable	Effect
<code>FCXXSS_BUILD_DIR=dir</code>	intermediate files (default <code>./build-fcxxss</code>)
<code>FCXXSS_NO_THREADS=1</code>	single-threaded build (<code>BOOST_DISABLE_THREADS</code>)
<code>FCXXSS_SHARED_PTR=1</code>	converted pointers are <code>boost::shared_node_ptr</code>
<code>FCXXSS_POINTER_TEMPLATE=X</code>	converted pointers are <code>X<T></code>
<code>FCXXSS_EXCEPTIONS=1</code>	build the runtime with exceptions
<code>FCXXSS_NO_STATIC_ASSERT=</code> (empty)	keep <code>static_assert</code> ; by default declarations are dropped so library assertions about trivially copyable types do not reject <code>root_ptr</code>
<code>FCXXSS_STDCXX=NN</code>	C++ standard: 17 (default), 20, 23, ...

11.4 Library macros

Pass these with `-D`; every file of a program must agree.

Macro	Effect
BOOST_ROOT_PTR_ALLOCATOR=A	node allocator template (default <code>boost::page_allocator_by_size</code>)
BOOST_DISABLE_THREADS	no locking (set by <code>FCXXSS_NO_THREADS=1</code>)
BOOST_PAGE_ALLOCATOR_THREAD_CACHE	per-thread free-block caches
BOOST_ROOT_PTR_STRIPED_LOCKS	striped locks for <code>root_ptr</code>
BOOST_ZEROIZATION	clear nodes before release
BOOST_NO_EXCEPTIONS	runtime never throws (the default)

Troubleshooting

License is ... or Concurrent processes.

See the messages in Chapter 4. `Concurrent processes.` under a trial or single licence means a parallel build: use `make` without `-j`.

unknown type name 'FCXXSS_RAW' (or another attribute)

The transformer did not see the macro. Include `fcxxss_attributes.h` (Chapter 7) in the file that uses it.

no matching function for call to a C function taking T **

A buffer of pointers is being passed to C. Mark it `FCXXSS_RAW` (Chapter 8).

new_into_system_owner_use_make_unique ... is unavailable

A `new` is handed to `std::unique_ptr` or `std::shared_ptr`. Use `std::make_unique` or `std::make_shared`.

Errors after changing flags or upgrading the Superset

Rebuild the precompiled header: `--clean`, or delete `.build-fcxxss`.

Link errors between files, or a class that differs between files

All files must be built with the same `FCXXSS_*` settings and `-D` macros, and all C++ files of the program must go through `fcxxss.sh`.

Inspecting what was generated

The transformed sources are in the build directory, under the absolute path of the original. They are ordinary C++ and can be compiled or read on their own.

12.1 Known limitations

- A cycle through `boost::shared_node_ptr` is never released — it is reference counted, like `std::shared_ptr`.
- `FCXXSS_RAW` on a typedef does not survive as a template argument: `std::vector<RawSlot>` is converted. The alias form `using A FCXXSS_RAW = void *;` is not recognised; use `typedef`.
- A raw pointer parameter is still wrapped in a `root_ptr` inside the function body, even with `FCXXSS_RAW_BODY`. That is harmless for `T *`, but a `T **` parameter is then read with the wrong layout. Write such functions in C (Section 8.4).
- `new` inside `sizeof(...)` is evaluated (`sizeof` of an expression becomes a function call).
- The runtime is built without exceptions by default (`BOOST_NO_EXCEPTIONS`): out-of-bounds accesses return a default instance instead of throwing.
- `static_assert` declarations are dropped unless `FCXXSS_NO_STATIC_ASSERT` is set to an empty value.

Legal notices

The Fornux C++ Superset is licensed under the Software Licence Agreement in `/opt/fornux/superset/LICENSE.md`. Individual runtime headers carry their own licence in their file header: `boost/smart_ptr/root_ptr.hpp` and the headers of its implementation are distributed under the Apache License, Version 2.0, and `fcxxss.hpp` under the Boost Software License, Version 1.0.

The method is protected by United States patent US11288049B2, *Source to source compiler, compilation method, and computer-readable medium for predictable memory management*.