# CXXR and Add-on Packages

Andrew Runnalls

School of Computing, University of Kent, UK

# Outline

# The CXXR Project

The aim of the CXXR project[1] is progressively to reengineer the fundamental parts of the R interpreter from C into C++.

By converting the interpreter internals to a well-documented object-oriented design, we hope that it will become easier for researchers to produce experimental versions of the interpreter, and explore new avenues for possible R development.

Work on CXXR started in May 2007, shadowing R-2.5.1; current work shadows R-2.10.1, with an upgrade to R-2.11.1 imminent.

We'll refer to the standard R interpreter as **CR**.

[1] www.cs.kent.ac.uk/projects/cxxr

# The CXXR Project

The aim of the CXXR project[1] is progressively to reengineer the fundamental parts of the R interpreter from C into C++.

By converting the interpreter internals to a well-documented object-oriented design, we hope that it will become easier for researchers to produce experimental versions of the interpreter, and explore new avenues for possible R development.

Work on CXXR started in May 2007, shadowing R-2.5.1; current work shadows R-2.10.1, with an upgrade to R-2.11.1 imminent.

We'll refer to the standard R interpreter as **CR**.

[1] www.cs.kent.ac.uk/projects/cxxr

# CXXR Constraints

At every stage of refactorization, CXXR aims to preserve the full functionality of the standard R distribution. In particular it is intended that as far as possible:

- The behaviour of R code is unaffected (unless it probes into the interpreter internals);
- The `.C`, `.Fortran`, `.Call` and `.External` call-out interfaces are unaffected;
- The `R.h` and `S.h` APIs are unaffected. (However, code compiled against `Rinternals.h` may need minor alterations.)

# Progress So Far

Important aspects of CXXR development to date include:

- The `SEXPREC` union has been replaced by an extensible hierarchy of C++ classes rooted at class `CXXR::RObject`. (All of CXXR's C++ code is placed within the C++ namespace `CXXR`, and we'll usually omit the prefix from now on.)
- Memory allocation and garbage collection have been completely refactored, and decoupled from R-specific functionality. Garbage collection is now based primarily on reference counting, with (non-generational) mark-sweep as a backstop.
- R's evaluation logic has been refactored into C++, with the exception so far of method dispatch.
- In a development branch, Chris Silles is providing facilities for tracking the provenance of R data objects (like the old `S AUDIT` facility), and for interrogating this provenance within a CXXR session.

## Progress So Far

Important aspects of CXXR development to date include:

- The `SEXPREC` union has been replaced by an extensible hierarchy of C++ classes rooted at class `CXXR::RObject`. (All of CXXR's C++ code is placed within the C++ namespace `CXXR`, and we'll usually omit the prefix from now on.)
- Memory allocation and garbage collection have been completely refactored, and decoupled from R-specific functionality. Garbage collection is now based primarily on reference counting, with (non-generational) mark-sweep as a backstop.
- R's evaluation logic has been refactored into C++, with the exception so far of method dispatch.
- In a development branch, Chris Silles is providing facilities for tracking the provenance of R data objects (like the old `S AUDIT` facility), and for interrogating this provenance within a CXXR session.

# Progress So Far

Important aspects of CXXR development to date include:

- The `SEXPREC` union has been replaced by an extensible hierarchy of C++ classes rooted at class `CXXR::RObject`. (All of CXXR's C++ code is placed within the C++ namespace `CXXR`, and we'll usually omit the prefix from now on.)
- Memory allocation and garbage collection have been completely refactored, and decoupled from R-specific functionality. Garbage collection is now based primarily on reference counting, with (non-generational) mark-sweep as a backstop.
- R's evaluation logic has been refactored into C++, with the exception so far of method dispatch.
- In a development branch, Chris Silles is providing facilities for tracking the provenance of R data objects (like the old `S AUDIT` facility), and for interrogating this provenance within a CXXR session.

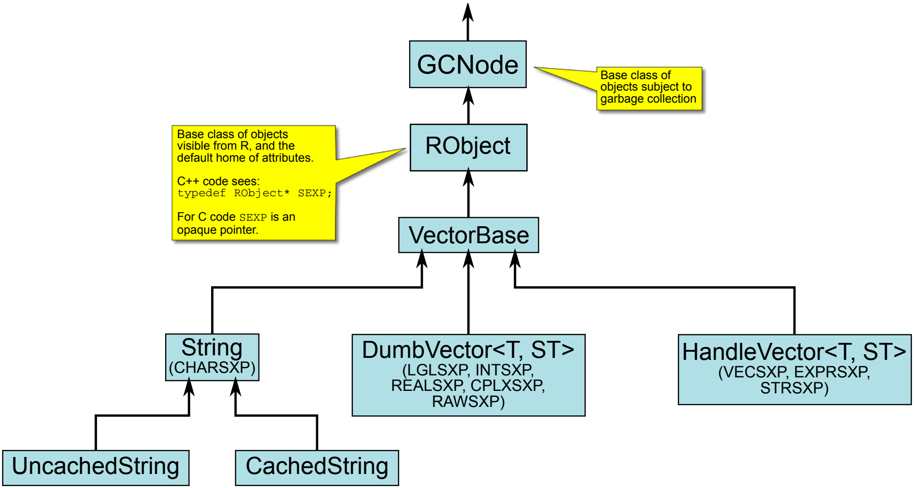
# Progress So Far

Important aspects of CXXR development to date include:

- The `SEXPREC` union has been replaced by an extensible hierarchy of C++ classes rooted at class `CXXR::RObject`. (All of CXXR's C++ code is placed within the C++ namespace `CXXR`, and we'll usually omit the prefix from now on.)
- Memory allocation and garbage collection have been completely refactored, and decoupled from R-specific functionality. Garbage collection is now based primarily on reference counting, with (non-generational) mark-sweep as a backstop.
- R's evaluation logic has been refactored into C++, with the exception so far of method dispatch.
- In a development branch, Chris Silles is providing facilities for tracking the provenance of R data objects (like the old `S AUDIT` facility), and for interrogating this provenance within a CXXR session.

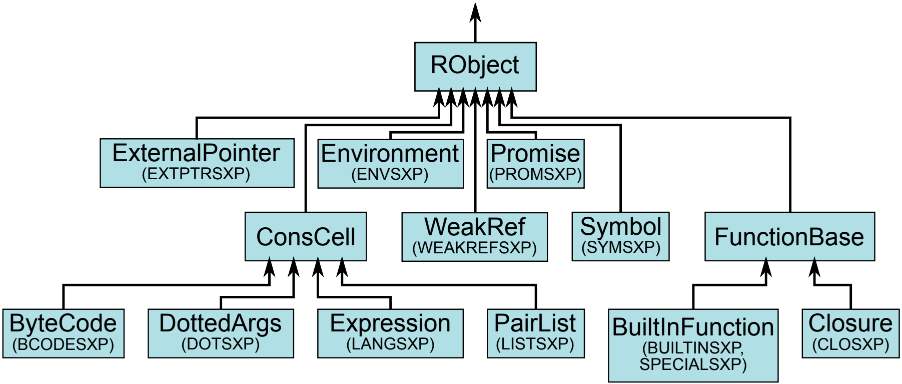
# The `RObject` Class Hierarchy

## Vector classes



# The `RObject` Class Hierarchy

## Other classes



- As far as possible, move all program code relating to a particular datatype into one place.
- Use C++'s public/protected/private mechanism to conceal implementational details and to defend class invariants, e.g.:
  - Every attribute of an `RObject` shall have a distinct `Symbol` object as its tag.
  - No two `Symbol` objects shall have the same name.
- Allow developers readily to extend the class hierarchy.

- As far as possible, move all program code relating to a particular datatype into one place.
- Use C++'s public/protected/private mechanism to conceal implementational details and to defend class invariants, e.g.:
  - Every attribute of an `RObject` shall have a distinct `Symbol` object as its tag.
  - No two `Symbol` objects shall have the same name.
- Allow developers readily to extend the class hierarchy.

# The `RObject` Class Hierarchy

## Objectives

- As far as possible, move all program code relating to a particular datatype into one place.
- Use C++'s public/protected/private mechanism to conceal implementational details and to defend class invariants, e.g.:
  - Every attribute of an `RObject` shall have a distinct `Symbol` object as its tag.
  - No two `Symbol` objects shall have the same name.
- Allow developers readily to extend the class hierarchy.

# Performance

The following tests were carried out on a 2.8 GHz Pentium 4 with 1 GB RAM and 1 MB L2 cache, comparing R-2.10.1 with CXXR release 0.29-2.10.1, using comparable optimization options. Times are CPU time (user + system).

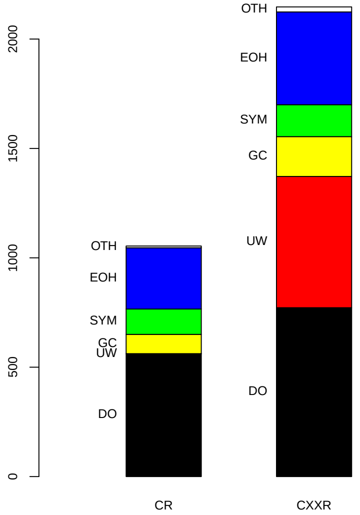
| **Benchmark** | **CR (secs)** | **CXXR (secs)** | **CR/CXXR** |
|---|---|---|---|
| `bench.R`[2] | 129.1 | 114.5 | 1.13 |
| `base5-Ex.R`[3] | 30.4 | 44.8 | 0.68 |
| `stats-Ex.R` | 48.7 | 92.7 | 0.53 |
| `jens.R`[4] | 116.2 | 78.1 | 1.49 |

[2] By Jan de Leeuw, at http://r.research.att.com/benchmarks.

[3] Fivefold concatenation of `base-Ex.R`, omitting internal `quit()`s.

[4] Based on example R code from Jens Oehlschlägel *Managing Large Datasets in R—ff Examples and Concepts* [2010].

# Timing Analysis with `stats-Ex.R`




- **DO** Time servicing `do_` functions, excluding nested R expression evaluation and the next three categories below.
- **UW** Stack unwinding, e.g. C++ exception propagation, or `findcontext()` in CR.
- **GC** Garbage collection.
- **SYM** Symbol look-up.
- **EOH** Evaluation overhead, i.e. time spent evaluating R expressions not included in the categories above.
- **OTH** Anything else, e.g. time spent outside the evaluation loop.

# Outline

# How Compatible is CXXR with Packages from CRAN?

Until this year, CXXR had only been tested with packages forming part of the standard distribution, including the ‘Recommended’ packages.

How well does it work with other packages from CRAN?

# How Compatible is CXXR with Packages from CRAN?

Until this year, CXXR had only been tested with packages forming part of the standard distribution, including the ‘Recommended’ packages.

How well does it work with other packages from CRAN?

We have now tried CXXR with 50 other packages from CRAN.

In choosing packages to test, we asked ‘How many other packages in CRAN depend on or suggest this package, directly or indirectly?’ The packages tested were those for which this was a maximum.

Many thanks to Uwe Ligges for a script to identify these packages.

# CRAN Packages Tested

| | | | | | |
|---|---|---|---|---|---|
| akima | 486 | DBI | 372 | fBasics | 318 |
| rgl | 472 | RSQLite | 360 | e1071 | 318 |
| RUnit | 463 | maps | 359 | ape | 318 |
| SparseM | 461 | mapproj | 359 | mix | 317 |
| RColorBrewer | 446 | robustbase | 354 | mclust | 316 |
| scatterplot3d | 438 | RODBC | 342 | leaps | 316 |
| mvtnorm | 416 | xtable | 337 | logspline | 315 |
| bitops | 397 | randomForest | 335 | quantreg | 313 |
| rsprng | 395 | gtools | 335 | numDeriv | 311 |
| rlecuyer | 393 | abind | 333 | multicore | 309 |
| Rmpi | 392 | tripack | 326 | mlmRev | 308 |
| nws | 391 | gee | 322 | lme4 | 308 |
| sp | 390 | timeDate | 321 | MEMSS | 308 |
| rpvm | 388 | biglm | 321 | slam | 304 |
| coda | 388 | timeSeries | 320 | kernlab | 303 |
| snow | 386 | mlbench | 318 | car | 302 |
| tkrplot | 384 | gdata | 318 | | |

Package versions were those current on 2010-05-05.

# Test Procedure

Working through the packages in order of decreasing number of reverse dependencies, each package was installed into CXXR with a (Unix shell) command such as:

```
> CXXR CMD INSTALL --install-tests foo_1.2-3.tar.gz
```

(For some packages a `--configure-args` flag was also necessary, and/or setting of environment variables.)

The package was then tested within a CXXR session with the R command:

```
> print(tools::testInstalledPackage('foo'))
```

This will carry out any package-specific tests as well as testing the package's code examples and vignettes.

How good a test this is varies enormously from package to package.

# Test Procedure

Working through the packages in order of decreasing number of reverse dependencies, each package was installed into CXXR with a (Unix shell) command such as:

```
> CXXR CMD INSTALL --install-tests foo_1.2-3.tar.gz
```

(For some packages a `--configure-args` flag was also necessary, and/or setting of environment variables.)

The package was then tested within a CXXR session with the R command:

```
> print(tools::testInstalledPackage('foo'))
```

This will carry out any package-specific tests as well as testing the package's code examples and vignettes.

How good a test this is varies enormously from package to package.

Of the 50 packages:

- 36 installed and tested OK 'out of the box'.
- A further two packages installed OK, and `testInstalledPackage` returned 0 (signifying OK) but under CXXR there were additional R warnings.
- In a further five packages, the test suite exhibited problems under both CXXR and our CR installation.[5] With appropriate tweaks and workarounds, three of these five packages then passed the tests under CXXR (and all of them under CR).

This makes a total of 41 packages that passed `testInstalledPackage` without altering either the package or CXXR.

[5] For example, there were three packages for which `testInstalledPackage` would only work if the working directory had the same name as the package.

Of the 50 packages:

- 36 installed and tested OK 'out of the box'.
- A further two packages installed OK, and `testInstalledPackage` returned 0 (signifying OK) but under CXXR there were additional R warnings.
- In a further five packages, the test suite exhibited problems under both CXXR and our CR installation.[5] With appropriate tweaks and workarounds, three of these five packages then passed the tests under CXXR (and all of them under CR).

This makes a total of 41 packages that passed `testInstalledPackage` without altering either the package or CXXR.

[5] For example, there were three packages for which `testInstalledPackage` would only work if the working directory had the same name as the package.

Of the 50 packages:

- 36 installed and tested OK 'out of the box'.
- A further two packages installed OK, and `testInstalledPackage` returned 0 (signifying OK) but under CXXR there were additional R warnings.
- In a further five packages, the test suite exhibited problems under both CXXR and our CR installation.[5] With appropriate tweaks and workarounds, three of these five packages then passed the tests under CXXR (and all of them under CR).

This makes a total of 41 packages that passed `testInstalledPackage` without altering either the package or CXXR.

[5] For example, there were three packages for which `testInstalledPackage` would only work if the working directory had the same name as the package.

Of the 50 packages:

- 36 installed and tested OK ‘out of the box’.
- A further two packages installed OK, and `testInstalledPackage` returned 0 (signifying OK) but under CXXR there were additional R warnings.
- In a further five packages, the test suite exhibited problems under both CXXR and our CR installation.[5] With appropriate tweaks and workarounds, three of these five packages then passed the tests under CXXR (and all of them under CR).

This makes a total of 41 packages that passed `testInstalledPackage` without altering either the package or CXXR.

[5] For example, there were three packages for which `testInstalledPackage` would only work if the working directory had the same name as the package.

# Results
## The not-so-good

- Five packages revealed bugs in CXXR (seven bugs in all). When these were fixed, all of them passed `testInstalledPackage`.
- Four packages proved to contain bugs (five bugs in all) that had remained latent under CR. In three cases, these were gaps in protection against garbage collection (i.e. missing `PROTECT()`/`UNPROTECT()`).
  After fixing these problems, three of the four packages then passed `testInstalledPackage`; the remaining package also fell foul of the next problem.
- Two packages included C code that was inconsistent with CXXR. Fixing these problems required changing three lines of code in all, and did not affect the packages' compatibility with CR.

After the changes described above, all 50 packages passed `testInstalledPackage`.

- Five packages revealed bugs in CXXR (seven bugs in all). When these were fixed, all of them passed `testInstalledPackage`.
- Four packages proved to contain bugs (five bugs in all) that had remained latent under CR. In three cases, these were gaps in protection against garbage collection (i.e. missing `PROTECT()`/`UNPROTECT()`).
  After fixing these problems, three of the four packages then passed `testInstalledPackage`; the remaining package also fell foul of the next problem.
- Two packages included C code that was inconsistent with CXXR. Fixing these problems required changing three lines of code in all, and did not affect the packages' compatibility with CR.

After the changes described above, all 50 packages passed `testInstalledPackage`.

# Results

## The not-so-good

- Five packages revealed bugs in CXXR (seven bugs in all). When these were fixed, all of them passed `testInstalledPackage`.
- Four packages proved to contain bugs (five bugs in all) that had remained latent under CR. In three cases, these were gaps in protection against garbage collection (i.e. missing `PROTECT()`/`UNPROTECT()`).
  After fixing these problems, three of the four packages then passed `testInstalledPackage`; the remaining package also fell foul of the next problem.
- Two packages included C code that was inconsistent with CXXR. Fixing these problems required changing three lines of code in all, and did not affect the packages' compatibility with CR.

After the changes described above, all 50 packages passed `testInstalledPackage`.

# Results

## The not-so-good

- Five packages revealed bugs in CXXR (seven bugs in all). When these were fixed, all of them passed `testInstalledPackage`.
- Four packages proved to contain bugs (five bugs in all) that had remained latent under CR. In three cases, these were gaps in protection against garbage collection (i.e. missing `PROTECT()`/`UNPROTECT()`).
  After fixing these problems, three of the four packages then passed `testInstalledPackage`; the remaining package also fell foul of the next problem.
- Two packages included C code that was inconsistent with CXXR. Fixing these problems required changing three lines of code in all, and did not affect the packages' compatibility with CR.

After the changes described above, all 50 packages passed `testInstalledPackage`.

# Outline

# Exploiting CXXR in Packages

**Extending the `RObject` hierarchy:** The internal `RObject` class hierarchy can be extended by packages, rather than their having to use external pointers and finalizers. This brings further benefits…

**'Virtual attributes':** C++ classes within the `RObject` hierarchy can apply their own checks on attribute settings, and determine how attribute values are stored within the class object.

**Delegated serialization/deserialization:** C++ classes within the `RObject` hierarchy can control how objects of that class are serialized. So custom objects can be saved as part of the CXXR session. (Work in progress.)

**Simpler GC-protection:** CR's `PROTECT()`/`REPROTECT()`/`UNPROTECT()` mechanism for protecting `SEXP`s against garbage collection is somewhat error prone. CXXR offers a much simpler mechanism using C++ smart pointers.

# Exploiting CXXR in Packages

**Extending the `RObject` hierarchy:** The internal `RObject` class hierarchy can be extended by packages, rather than their having to use external pointers and finalizers. This brings further benefits. . .

**'Virtual attributes':** C++ classes within the `RObject` hierarchy can apply their own checks on attribute settings, and determine how attribute values are stored within the class object.

**Delegated serialization/deserialization:** C++ classes within the `RObject` hierarchy can control how objects of that class are serialized. So custom objects can be saved as part of the CXXR session. (Work in progress.)

**Simpler GC-protection:** CR's `PROTECT()/REPROTECT()/UNPROTECT()` mechanism for protecting `SEXP`s against garbage collection is somewhat error prone. CXXR offers a much simpler mechanism using C++ smart pointers.

# Exploiting CXXR in Packages

**Extending the `RObject` hierarchy:** The internal `RObject` class hierarchy can be extended by packages, rather than their having to use external pointers and finalizers. This brings further benefits. . .

**'Virtual attributes':** C++ classes within the `RObject` hierarchy can apply their own checks on attribute settings, and determine how attribute values are stored within the class object.

**Delegated serialization/deserialization:** C++ classes within the `RObject` hierarchy can control how objects of that class are serialized. So custom objects can be saved as part of the CXXR session. (Work in progress.)

**Simpler GC-protection:** CR's `PROTECT()`/`REPROTECT()`/`UNPROTECT()` mechanism for protecting `SEXP`s against garbage collection is somewhat error prone. CXXR offers a much simpler mechanism using C++ smart pointers.

# Exploiting CXXR in Packages

**Extending the `RObject` hierarchy:** The internal `RObject` class hierarchy can be extended by packages, rather than their having to use external pointers and finalizers. This brings further benefits. . .

**'Virtual attributes':** C++ classes within the `RObject` hierarchy can apply their own checks on attribute settings, and determine how attribute values are stored within the class object.

**Delegated serialization/deserialization:** C++ classes within the `RObject` hierarchy can control how objects of that class are serialized. So custom objects can be saved as part of the CXXR session. (Work in progress.)

**Simpler GC-protection:** CR's `PROTECT()`/`REPROTECT()`/`UNPROTECT()` mechanism for protecting `SEXP`s against garbage collection is somewhat error prone. CXXR offers a much simpler mechanism using C++ smart pointers.

# Outline

# Next Stages

- Upgrade CXXR to shadow R 2.11.1
- Port CXXR to Windows. Any volunteers?
- Improve performance.
- At present data provenance is tracked only within a single R session. This is being extended to cross-session tracking.
- Refactor method-dispatch code into C++.
- Consider how better to handle R's array subscripting/subsetting operations within a C++ framework. The present `VectorBase` class is underpowered, and does not provide a mature base for CXXR package-writers to build on.

# Performance

The following tests were carried out on a 2.8 GHz Pentium 4 with 1 GB RAM and 1 MB L2 cache, comparing R-2.10.1 with CXXR release 0.29-2.10.1, using comparable optimization options. Times are CPU time (user + system).

| **Benchmark** | **CR (secs)** | **CXXR (secs)** | **CR/CXXR** |
|---|---|---|---|
| `bench.R`[6] | $129.1 \pm 0.4$ | $114.5 \pm 0.2$ | 1.13 |
| `base5-Ex.R`[7] | $30.4 \pm 0.1$ | $44.8 \pm 0.7$ | 0.68 |
| `stats-Ex.R` | $48.7 \pm 0.1$ | $92.7 \pm 0.4$ | 0.53 |
| `jens.R`[8] | $116.2 \pm 0.3$ | $78.1 \pm 0.7$ | 1.49 |

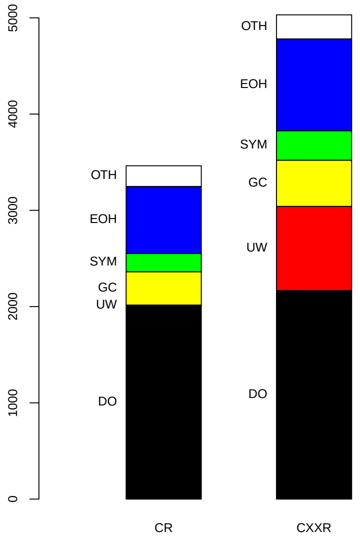
(Means of 5 runs; tolerances $2\sigma$)

[6] By Jan de Leeuw, at http://r.research.att.com/benchmarks.

[7] Fivefold concatenation of `base-Ex.R`, omitting internal `quit()`s.

[8] Based on example R code from Jens Oehlschlägel *Managing Large Datasets in R—ff Examples and Concepts* [2010].

# Timing Analysis with `base5-Ex.R`




DO Time servicing `do_` functions, excluding nested R expression evaluation and the next three categories below.

UW Stack unwinding, e.g. C++ exception propagation, or `findcontext()` in CR.

GC Garbage collection.

SYM Symbol look-up.

EOH Evaluation overhead, i.e. time spent evaluating R expressions not included in the categories above.

OTH Anything else, e.g. time spent outside the evaluation loop.

# Nested LCONS

In CXXR, objects of type LANGSXP (implemented by C++ class Expression), DOTSXP (class DottedArgs) and BCODESXP (class ByteCode) are permitted only to appear at the head of a pairlist; all remaining elements of the list must be of type LISTSXP (class PairList).

So for example the C code:

```
SEXP hcall = LCONS(h, LCONS(cond, R_NilValue));
```

needs to be changed to

```
SEXP hcall = LCONS(h, CONS(cond, R_NilValue));
```

for use under CXXR.

# Code Migration from R to C++

In CXXR, underlying every R object (whether of an R class type or not) is a C++ object of a class inheriting from `RObject`.

Very often in R packages, much code is specifically associated with a particular type of R object. This is most obvious in R class definitions. The code in question may be written in R itself, in C or C++, or maybe in some other language.

CXXR aims to allow you easily to migrate the functionality of that code into the C++ class underlying those objects. This can be done in small steps, and to the extent that you see fit.

# Code Migration from R to C++

In CXXR, underlying every R object (whether of an R class type or not) is a C++ object of a class inheriting from `RObject`.

Very often in R packages, much code is specifically associated with a particular type of R object. This is most obvious in R class definitions. The code in question may be written in R itself, in C or C++, or maybe in some other language.

CXXR aims to allow you easily to migrate the functionality of that code into the C++ class underlying those objects. This can be done in small steps, and to the extent that you see fit.

# Code Migration from R to C++

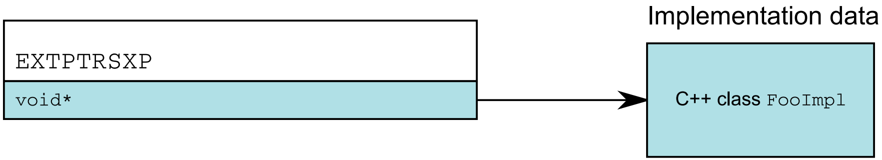
In CXXR, underlying every R object (whether of an R class type or not) is a C++ object of a class inheriting from `RObject`.

Very often in R packages, much code is specifically associated with a particular type of R object. This is most obvious in R class definitions. The code in question may be written in R itself, in C or C++, or maybe in some other language.

CXXR aims to allow you easily to migrate the functionality of that code into the C++ class underlying those objects. This can be done in small steps, and to the extent that you see fit.

# Evolution of an R Class under CXXR

## R Class "foo"

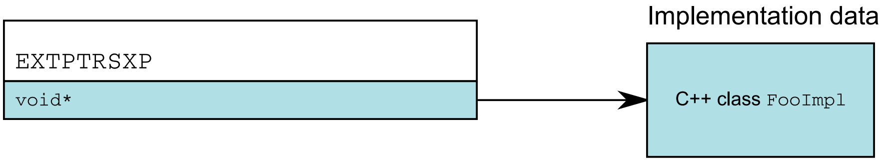



An ‘external pointer’ R object contains an untyped pointer, which can be configured to point to an arbitrary C/C++ data structure.

A common issue is how to recover the memory space used by this data structure when the external pointer object is garbage-collected. The standard approach is to use a **finalizer**. . .

# Evolution of an R Class under CXXR

## R Class `"foo"`

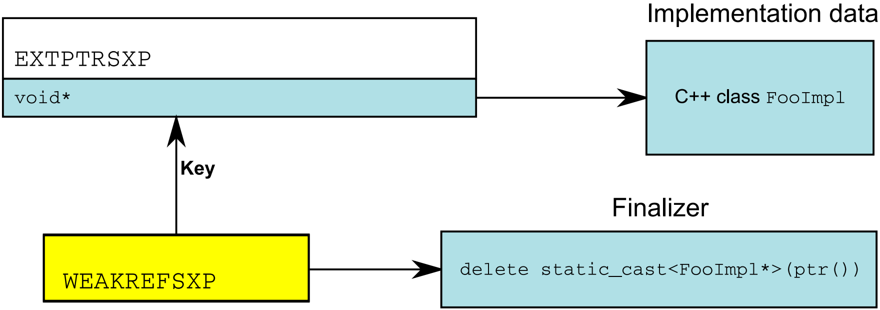



An ‘external pointer’ R object contains an untyped pointer, which can be configured to point to an arbitrary C/C++ data structure.

A common issue is how to recover the memory space used by this data structure when the external pointer object is garbage-collected. The standard approach is to use a **finalizer**…

# Evolution of an R Class under CXXR

## R Class "foo"

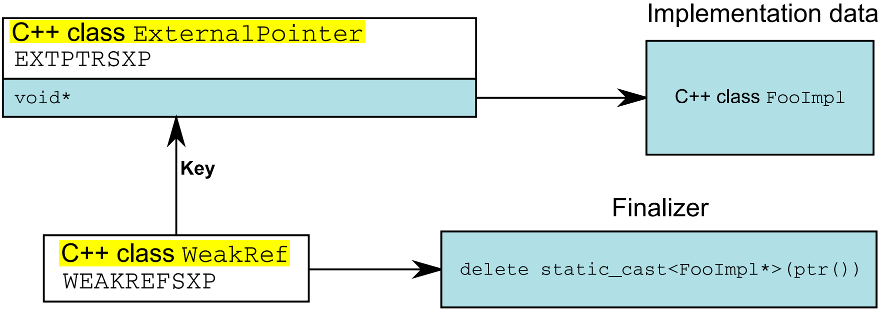



Finalization is implemented using an auxiliary ‘weak reference’ object, which designates the object to be finalized as its **key**.

During a mark-sweep garbage collection, if it is determined that the key of a weak reference is unreachable, the finalizer is executed. Then the key and the weak reference are garbage-collected.

# Evolution of an R Class under CXXR

R Class `"foo"`

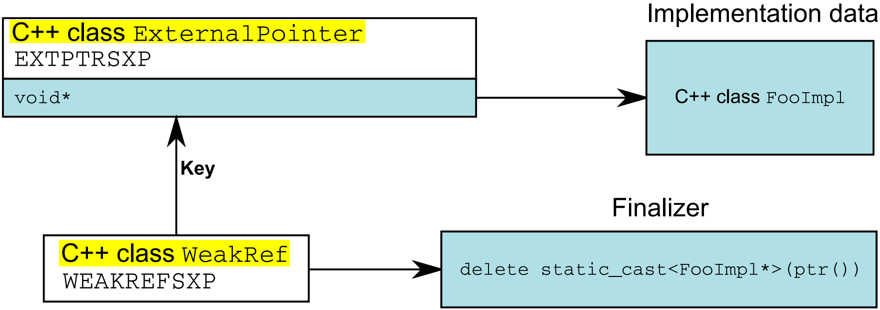



The same mechanism remains available under CXXR, implemented via the class `WeakRef`.

A drawback is that neither weak reference objects nor their keys can be garbage-collected by the reference counting scheme. Consequently objects of class `"foo"` will remain in existence until the next full garbage collection.

# Evolution of an R Class under CXXR

## R Class "foo"

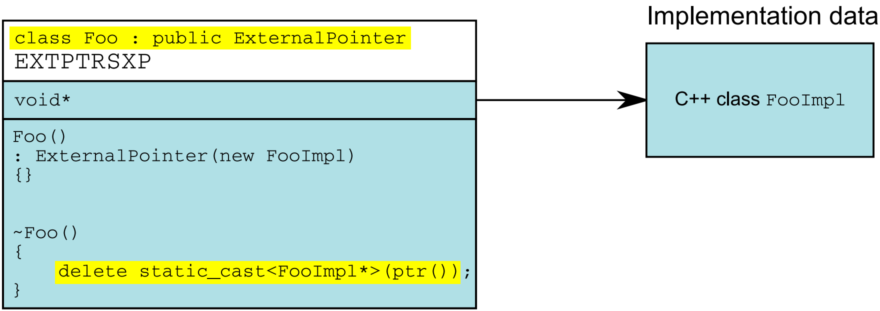



The same mechanism remains available under CXXR, implemented via the class `WeakRef`.

A drawback is that neither weak reference objects nor their keys can be garbage-collected by the reference counting scheme. Consequently objects of class `"foo"` will remain in existence until the next full garbage collection.

# Evolution of an R Class under CXXR

## R Class "foo"




An easy change: instead of using class `ExternalPointer` itself, we can introduce a new C++ class `Foo` inheriting from `ExternalPointer`, and incorporate the finalization logic in the class destructor. `Foo` objects can now be garbage-collected by reference counting.

## R Class "foo"

```
class Foo : public VectorBase

FooImpl* m_impl;

Foo(size_t size)
: VectorBase(size),
  m_impl(new FooImpl(size))
{}

~Foo()
{
    delete m_impl;
}
```

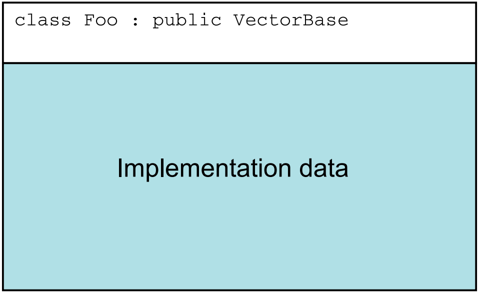
Implementation data

C++ class `FooImpl`

But why use `ExternalPointer` objects at all? If, for example, class `"foo"` has the characteristics of a data vector, we can make its C++ representation inherit instead from `VectorBase`.

# Evolution of an R Class under CXXR

## R Class "`foo`"




Finally, we may be able to incorporate the C++ data structures implementing class "`foo`" directly into the `Foo` object, eliminating an indirection and probably simplifying the code.

# Attributes in CR

Each R object can have a list of named attributes associated with it.

Under CR, the C function `setAttrib()` applies checks to the value supplied for any attribute named `"class"`, `"comment"`, `"dim"`, `"dimnames"`, `"names"`, `"row.names"` or `"tsp"`. Apart from that, anything goes.

# CXXR: Virtual Attributes

In CXXR, the trend is to delegate attribute control to individual classes within the `RObject` hierarchy.

Class `RObject` contains a pairlist of attributes just as in CR. Attribute values are set using the method:

```
virtual void RObject::setAttribute(const Symbol* name,
                                   RObject* value);
```

However, because this method (and other attribute-related methods) are declared `virtual`, their default implementations can be *overridden* by other C++ classes in the `RObject` hierarchy.

# CXXR: Virtual Attributes

In CXXR, the trend is to delegate attribute control to individual classes within the `RObject` hierarchy.

Class `RObject` contains a pairlist of attributes just as in CR. Attribute values are set using the method:

```
virtual void RObject::setAttribute(const Symbol* name,
                                   RObject* value);
```

However, because this method (and other attribute-related methods) are declared `virtual`, their default implementations can be *overridden* by other C++ classes in the `RObject` hierarchy.

# CXXR: Virtual Attributes

Where CXXR packages provide new C++ classes within the `RObject` hierarchy, they can use this 'virtual attribute' facility in two ways:

- To apply class-specific checks that attribute values are consistent with the C++ class invariants. For example, arrays from package `ff` have a `"dimorder"` attribute which determines their layout (row-major, column-major etc.). The underlying C++ class could verify that any value supplied for this attribute is a permutation of $1 : n$.
- To use an internal representation of attribute values that augments or replaces the default representation. For example, the value of a `"rotation"` attribute may appear to the R user to be an angle but be stored internally as a sine/cosine matrix.

# CXXR: Virtual Attributes

Where CXXR packages provide new C++ classes within the `RObject` hierarchy, they can use this ‘virtual attribute’ facility in two ways:

- To apply class-specific checks that attribute values are consistent with the C++ class invariants. For example, arrays from package `ff` have a `"dimorder"` attribute which determines their layout (row-major, column-major etc.). The underlying C++ class could verify that any value supplied for this attribute is a permutation of $1:n$.
- To use an internal representation of attribute values that augments or replaces the default representation. For example, the value of a `"rotation"` attribute may appear to the R user to be an angle but be stored internally as a sine/cosine matrix.

# Delegated Serialization/Deserialization

## (Work in progress: early days!)

Being able to track data object provenance from one R session to another means that information about the provenance of data objects must be saved alongside the data objects themselves. This is leading to a revision to the way in which object serialization and deserialization are carried out in CXXR.

As part of this, serialization and deserialization will be carried out by virtual functions of the abstract class `CXXR::Serializable`, from which `CXXR::RObject` will inherit.

CXXR package-writers who augment the `RObject` class hierarchy will be able to exploit this to save and restore their custom objects between CXXR sessions.

# Delegated Serialization/Deserialization

(Work in progress: early days!)

Being able to track data object provenance from one R session to another means that information about the provenance of data objects must be saved alongside the data objects themselves. This is leading to a revision to the way in which object serialization and deserialization are carried out in CXXR.

As part of this, serialization and deserialization will be carried out by virtual functions of the abstract class `CXXR::Serializable`, from which `CXXR::RObject` will inherit.

CXXR package-writers who augment the `RObject` class hierarchy will be able to exploit this to save and restore their custom objects between CXXR sessions.

# GC Protection Using Smart Pointers

The following example gives the flavour of C++ programming for CXXR:

```
// Return a reversed copy of a pairlist:

PairList* reverse(const PairList* inlist)
{
    GCStackRoot<PairList> revlist;
    while (inlist) {
        revlist
            = PairList::construct(inlist->car(), revlist,
                                  inlist->tag());
        inlist = inlist->tail();
    }
    return RObject::clone(revlist);
}
```

# GC Protection Using Smart Pointers

The following example gives the flavour of C++ programming for CXXR:

```
// Return a reversed copy of a pairlist:

PairList* reverse(const PairList* inlist)
{
    GCStackRoot<PairList> revlist;
    while (inlist) {
        revlist
                    t(inlist->car(), revlist,
                      inlist->tag());

    }
    re              t);
}
```

`GCStackRoot` **is a (templated) 'smart pointer' type. It can be used like a pointer (`PairList*` in this case) but protects whatever it points to from garbage collection.**

# GC Protection Using Smart Pointers

The following example gives the flavour of C++ programming for CXXR:

```
// Return a reversed copy of a pairlist:

PairList* reverse(PairList* inlist)
{
    GCStackRoot<PairList> revlist;
    while (inlist) {
        revlist
            = PairList::construct(inlist->car(), revlist,
                                  inlist->tag());
        inlist = inlist->tail();
    }
    return RObject::clone(revlist);
}
```

No need for `REPROTECT()` here.

# GC Protection Using Smart Pointers

The following example gives the flavour of C++ programming for CXXR:

```
// Return a reversed copy of a pairlist:

PairList* reverse(const PairList* inlist)
{
    GCStackRoot<PairList> revlist;
    while (inlist) {
        revlist
            = PairList::construct(inlist->car(), revlist,
                                  inlist->tag());
        inlist = inlist->tail();
    }
    return RObject::clone(revlist);
}
```

The `revlist` smart pointer goes out of scope here, and its destructor automatically ends the GC protection it offers. No need for `UNPROTECT()`.

# GC Protection Using Smart Pointers

The following example gives the flavour of C++ programming for CXXR:

```
// Return a reversed copy of a pairlist:

PairList* reverse(const PairList* inlist)
{
    GCStackRoot<PairList> revlist;
    while (inlist) {
        revlist
            = PairList::construct(inlist->car(), revlist,
                                  inlist->tag());
        inlist = inlist->tail();
    }
    return RObject::clone(revlist);
}
```

But if you prefer to do things the CR way, CXXR permits that too!