Aspects of CXXR Internals

Andrew Runnalls

Computing Laboratory, University of Kent, UK

Outline

1. CXXR
2. The `RObject` Class Hierarchy
3. Memory Allocation and Garbage Collection
4. Object Duplication
5. Environments
6. Performance
7. Looking Forward

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++, with the intention that:

- Full functionality of the standard R distribution is preserved;
- The behaviour of R code is unaffected (unless it probes into the interpreter internals);
- The `.C` and `.Fortran` interfaces, and the `R.h` and `S.h` APIs, are unaffected;
- Code compiled against `Rinternals.h` may need minor alterations.

Work started in May 2007, shadowing R-2.5.1; current work shadows R-2.8.1.

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++, with the intention that:

- Full functionality of the standard R distribution is preserved;
- The behaviour of R code is unaffected (unless it probes into the interpreter internals);
- The `.C` and `.Fortran` interfaces, and the `R.h` and `S.h` APIs, are unaffected;
- Code compiled against `Rinternals.h` may need minor alterations.

Work started in May 2007, shadowing R-2.5.1; current work shadows R-2.8.1.

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

---

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

Why Do This?

The medium-term objective is to introduce provenance-tracking facilities into CXXR: so that for any R data object, it is possible to determine exactly which original data files it was produced from, and exactly which sequence of operations was used to produce it. (Similar to the old S AUDIT facility, but usable directly within R.) Chris Silles made a presentation on this at useR! 2009.

Also:

- By improving the internal documentation, and
- Tightening up the internal encapsulation boundaries within the interpreter,

we hope that CXXR will make it easier for other researchers to produce experimental versions of the interpreter, and to enhance its facilities.

Why Do This?

The medium-term objective is to introduce provenance-tracking facilities into CXXR: so that for any R data object, it is possible to determine exactly which original data files it was produced from, and exactly which sequence of operations was used to produce it. (Similar to the old S AUDIT facility, but usable directly within R.) Chris Silles made a presentation on this at useR! 2009.

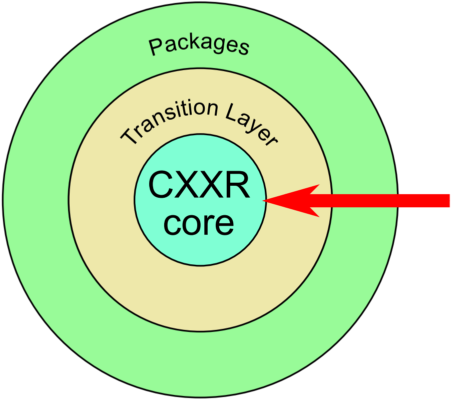
Also:

- By improving the internal documentation, and
- Tightening up the internal encapsulation boundaries within the interpreter,

we hope that CXXR will make it easier for other researchers to produce experimental versions of the interpreter, and to enhance its facilities.

CXXR Layers

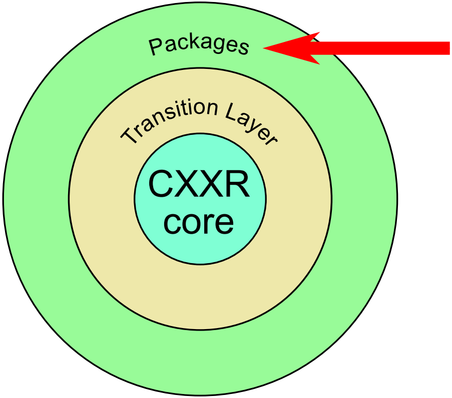
Core



- Written as far as possible in idiomatic C++, making free use of the C++ standard library (including some TR1 classes).
- Contained in the C++ namespace `CXXR`.
- Interfaces thoroughly documented using doxygen.
- As far as possible self-contained: avoids calls into the outer layers.

CXXR Layers

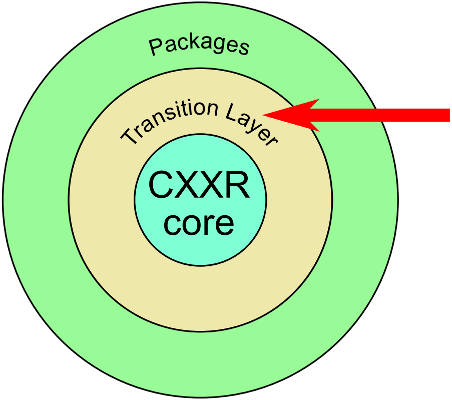
Packages



- Very few changes have proved necessary, e.g. only 9 `.c` files under `src/library` changed, with 46 changes in all. (This will shortly reduce to about 26 changes in 5 files.)
- But only the standard packages have been tested.

CXXR Layers

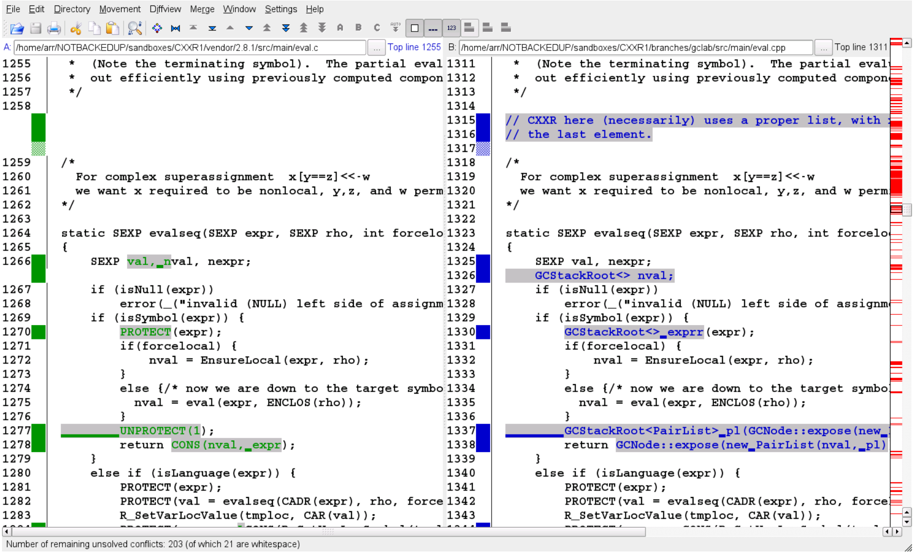
Transition layer



- CR files from `src/main` adapted as necessary to work with the core.
- With a few exceptions, C files have been redesignated as C++ (but CR idioms largely retained).
- Generally uses C linkage conventions, i.e. function names are not mangled to include information about argument types (as is the C++ default).
- Some special constructs used to facilitate upgrading to new releases of CR.

Example File from the Transition Layer

CR's `eval.c` *vs* CXXR's `eval.cpp`



A: /home/arr/NOTBACKEDUP/sandboxes/CXXR1/vendor/2.8.1/src/main/eval.c Top line 1255

```
 *  (Note the terminating symbol).  The partial eval
 *  out efficiently using previously computed compon
 */


/*
  For complex superassignment  x[y==z]<<-w
  we want x required to be nonlocal, y,z, and w perm
*/

static SEXP evalseq(SEXP expr, SEXP rho, int forcelo
{
    SEXP val, nval, nexpr;

    if (isNull(expr))
        error(_("invalid (NULL) left side of assignm
    if (isSymbol(expr)) {
        PROTECT(expr);
        if(forcelocal) {
            nval = EnsureLocal(expr, rho);
        }
        else {/* now we are down to the target symbo
          nval = eval(expr, ENCLOS(rho));
        }
        UNPROTECT(1);
        return CONS(nval, expr);
    }
    else if (isLanguage(expr)) {
        PROTECT(expr);
        PROTECT(val = evalseq(CADR(expr), rho, force
        R_SetVarLocValue(tmploc, CAR(val));
```

B: /home/arr/NOTBACKEDUP/sandboxes/CXXR1/branches/gclab/src/main/eval.cpp Top line 1311

```
 *  (Note the terminating symbol).  The partial eval
 *  out efficiently using previously computed compon
 */

// CXXR here (necessarily) uses a proper list, with
// the last element.

/*
  For complex superassignment  x[y==z]<<-w
  we want x required to be nonlocal, y,z, and w perm
*/

static SEXP evalseq(SEXP expr, SEXP rho, int forcelo
{
    SEXP val, nexpr;
    GCStackRoot<> nval;
    if (isNull(expr))
        error(_("invalid (NULL) left side of assignm
    if (isSymbol(expr)) {
        GCStackRoot<> exprr(expr);
        if(forcelocal) {
            nval = EnsureLocal(expr, rho);
        }
        else {/* now we are down to the target symbo
          nval = eval(expr, ENCLOS(rho));
        }
        GCStackRoot<PairList> pl(GCNode::expose(new_
        return GCNode::expose(new_PairList(nval, pl))
    }
    else if (isLanguage(expr)) {
        PROTECT(expr);
        PROTECT(val = evalseq(CADR(expr), rho, force
        R_SetVarLocValue(tmploc, CAR(val));
```

Number of remaining unsolved conflicts: 203 (of which 21 are whitespace)

`uncxxr.pl`

CXXR makes many systematic changes to files in the transition layer. For example:

- CR files are apt to use C++ reserved words (e.g. `this`, `new`, `class`) as identifiers. These have to be changed.
- C++ requires explicit conversions in places where C doesn't, e.g. `int` to enumeration, or `void*` to other pointer types.
- CXXR everywhere replaces C-style casts by C++ casts (e.g. `static_cast`, `const_cast`, `reinterpret_cast`).

Increasingly these (and other) changes are carried out in such a way that they can easily be reversed by a Perl script `uncxxr.pl`. This considerably eases the task of upgrading CXXR for a new release of CR.

`uncxxr.pl`

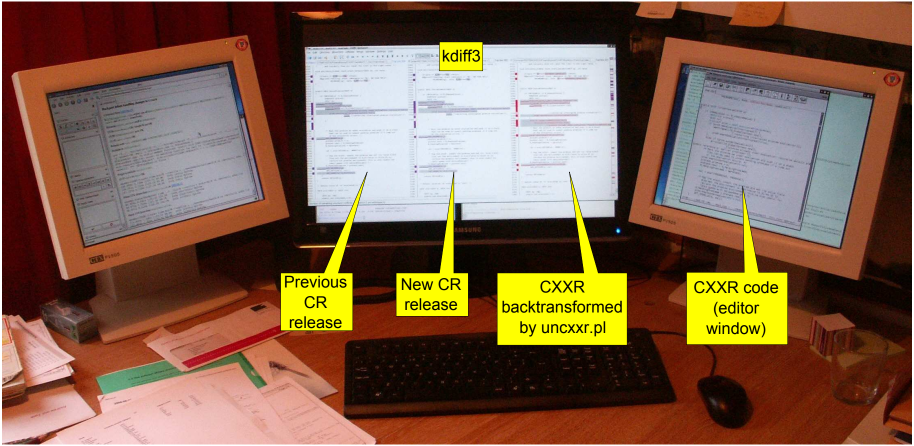
CXXR makes many systematic changes to files in the transition layer. For example:

- CR files are apt to use C++ reserved words (e.g. `this`, `new`, `class`) as identifiers. These have to be changed.
- C++ requires explicit conversions in places where C doesn't, e.g. `int` to enumeration, or `void*` to other pointer types.
- CXXR everywhere replaces C-style casts by C++ casts (e.g. `static_cast`, `const_cast`, `reinterpret_cast`).

Increasingly these (and other) changes are carried out in such a way that they can easily be reversed by a Perl script `uncxxr.pl`. This considerably eases the task of upgrading CXXR for a new release of CR.

Transition Layer

Updating to a new CR release



Functionality Now in CXXR Core

- Memory allocation and garbage collection.
- `SEXPREC` union replaced by an extensible class hierarchy.
- Object duplication (now handled essentially by C++ copy constructors).
- Environments (i.e. variable→object mappings), with hooks to support provenance tracking.

Outline

1. CXXR
2. The `RObject` Class Hierarchy
3. Memory Allocation and Garbage Collection
4. Object Duplication
5. Environments
6. Performance
7. Looking Forward

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. (See particularly the documentation of class `GCNode` for guidelines.)

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. (See particularly the documentation of class `GCNode` for guidelines.)

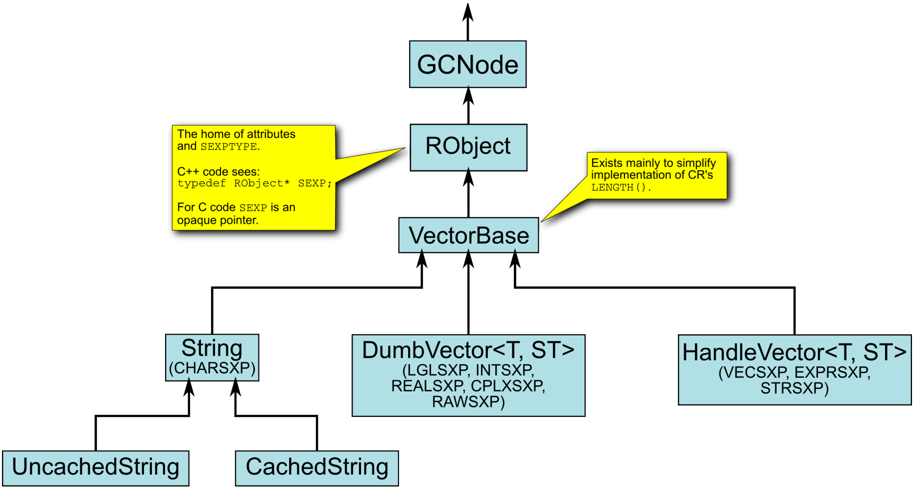
The `RObject` Class Hierarchy

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. (See particularly the documentation of class `GCNode` for guidelines.)

The `RObject` Class Hierarchy

Vector classes



The home of attributes and `SEXPTYPE`.

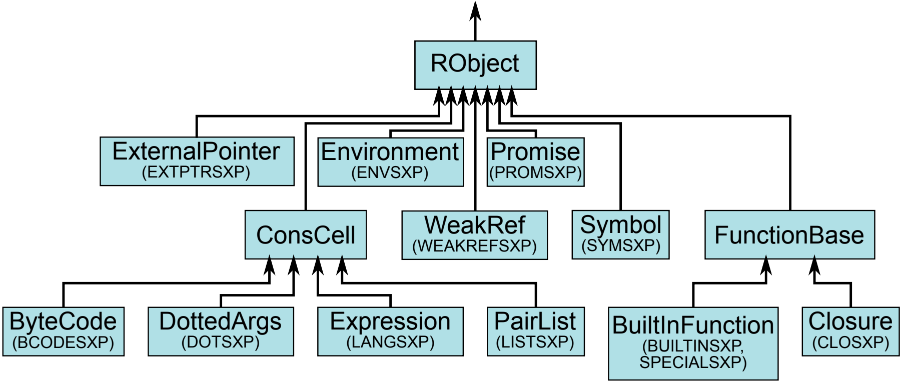
C++ code sees:
`typedef RObject* SEXP;`

For C code `SEXP` is an opaque pointer.

Exists mainly to simplify implementation of CR's `LENGTH()`.

The `RObject` Class Hierarchy

Other classes



RObject

ExternalPointer (EXTPTRSXP)

Environment (ENVSXP)

Promise (PROMSXP)

ConsCell

WeakRef (WEAKREFSXP)

Symbol (SYMSXP)

FunctionBase

ByteCode (BCODESXP)

DottedArgs (DOTSXP)

Expression (LANGSXP)

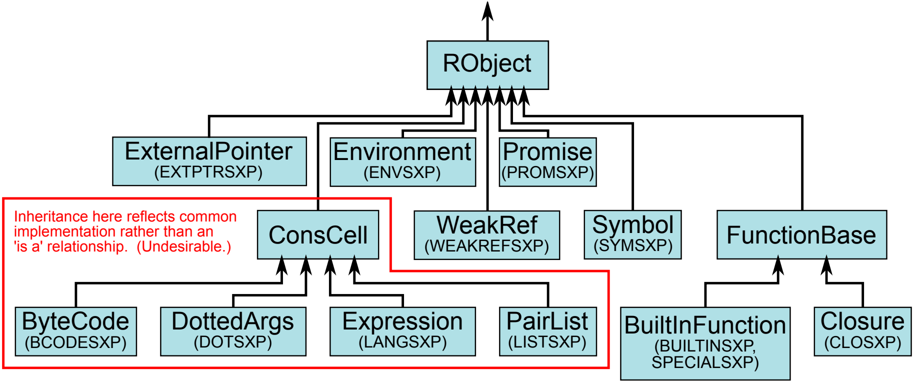
PairList (LISTSXP)

BuiltInFunction (BUILTINSXP, SPECIALSXP)

Closure (CLOSXP)

The `RObject` Class Hierarchy

Other classes



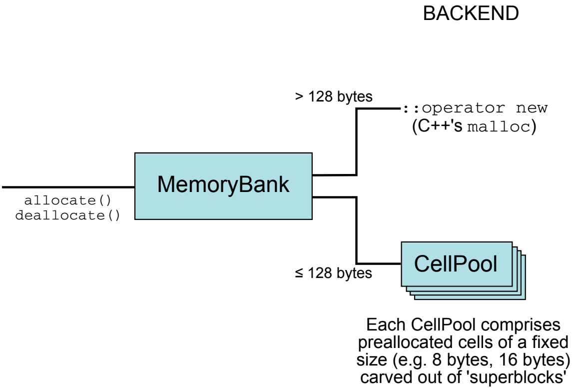
Inheritance here reflects common implementation rather than an 'is a' relationship. (Undesirable.)

Outline

1. CXXR
2. The `RObject` Class Hierarchy
3. **Memory Allocation and Garbage Collection**
4. Object Duplication
5. Environments
6. Performance
7. Looking Forward

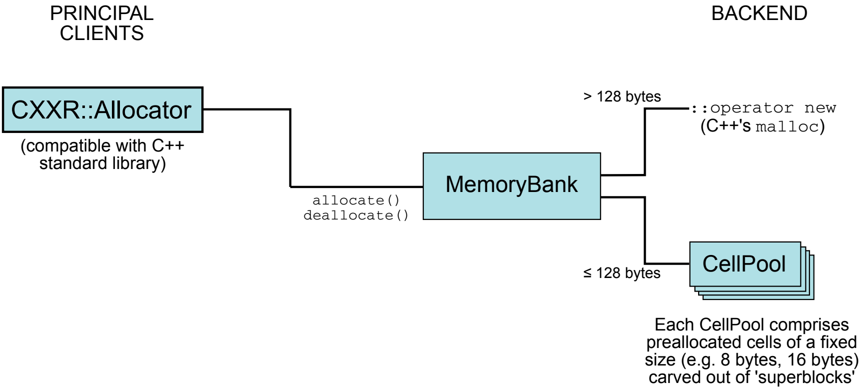
Memory Allocation

Memory allocation in CXXR is managed by the class `MemoryBank`.



Memory Allocation

Memory allocation in CXXR is managed by the class `MemoryBank`.



PRINCIPAL
CLIENTS

BACKEND

CXXR::Allocator

(compatible with C++
standard library)

`allocate()`
`deallocate()`

MemoryBank

> 128 bytes

`::operator new`
(C++'s `malloc`)

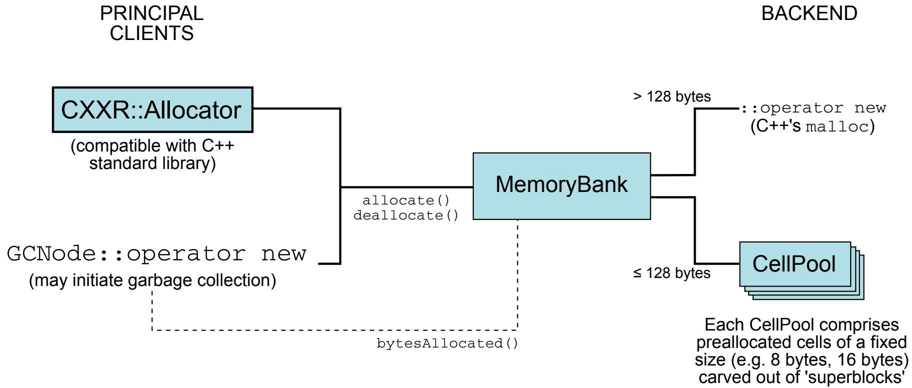
≤ 128 bytes

CellPool

Each CellPool comprises
preallocated cells of a fixed
size (e.g. 8 bytes, 16 bytes)
carved out of 'superblocks'

Memory Allocation

Memory allocation in CXXR is managed by the class `MemoryBank`.



PRINCIPAL CLIENTS

BACKEND

CXXR::Allocator

(compatible with C++ standard library)

`GCNode::operator new`

(may initiate garbage collection)

`allocate()`
`deallocate()`

MemoryBank

`bytesAllocated()`

> 128 bytes

`::operator new` (C++'s `malloc`)

≤ 128 bytes

CellPool

Each CellPool comprises preallocated cells of a fixed size (e.g. 8 bytes, 16 bytes) carved out of 'superblocks'

Garbage Collection Approaches

`trunk`	Generational mark-sweep (much as in CR).
`branches/refcount+gen`	Reference counting backed up by generational mark-sweep. (An intermediate refactorisation step.)
`branches/gclab`	Reference counting backed up by non-generational mark-sweep.

Garbage Collection Approaches

`trunk`	Generational mark-sweep (much as in CR).
`branches/refcount+gen`	Reference counting backed up by generational mark-sweep. (An intermediate refactorisation step.)
`branches/gclab`	Reference counting backed up by non-generational mark-sweep. Shortly to move to `trunk`: subsequent slides refer to this approach unless otherwise stated.

Garbage Collection: Principal Classes

`GCNode`	Base class for all objects subject to automatic garbage collection. A `GCNode` object incorporates a 1-byte saturating reference count: if the reference count ever gets to 255, it sticks there.
`GCEdge<T>`	A templated 'smart pointer' type, which `GCNode` objects use to refer to other `GCNode`s. Automatically adjusts the reference count of the node referred to. (In approaches using generational mark-sweep, it encapsulates the write barrier.)
`GCManager`	Controls the threshold(s) at which mark-sweep garbage collection will take place. (In approaches using generational mark-sweep, it also controls how many generations are collected.)
`GCRoot<T>` and `GCStackRoot<T>`	Smart pointer types used to provide long- and short-term protection against garbage collection.

Garbage Collection: Principal Classes

`GCNode`	Base class for all objects subject to automatic garbage collection. A `GCNode` object incorporates a 1-byte saturating reference count: if the reference count ever gets to 255, it sticks there.
`GCEdge<T>`	A templated ‘smart pointer’ type, which `GCNode` objects use to refer to other `GCNode`s. Automatically adjusts the reference count of the node referred to. (In approaches using generational mark-sweep, it encapsulates the write barrier.)
`GCManager`	Controls the threshold(s) at which mark-sweep garbage collection will take place. (In approaches using generational mark-sweep, it also controls how many generations are collected.)
`GCRoot<T>` and `GCStackRoot<T>`	Smart pointer types used to provide long- and short-term protection against garbage collection.

Garbage Collection: Principal Classes

`GCNode`	Base class for all objects subject to automatic garbage collection. A `GCNode` object incorporates a 1-byte saturating reference count: if the reference count ever gets to 255, it sticks there.
`GCEdge<T>`	A templated 'smart pointer' type, which `GCNode` objects use to refer to other `GCNode`s. Automatically adjusts the reference count of the node referred to. (In approaches using generational mark-sweep, it encapsulates the write barrier.)
`GCManager`	Controls the threshold(s) at which mark-sweep garbage collection will take place. (In approaches using generational mark-sweep, it also controls how many generations are collected.)
`GCRoot<T>` and `GCStackRoot<T>`	Smart pointer types used to provide long- and short-term protection against garbage collection.

Garbage Collection: Principal Classes

`GCNode`	Base class for all objects subject to automatic garbage collection. A `GCNode` object incorporates a 1-byte saturating reference count: if the reference count ever gets to 255, it sticks there.
`GCEdge<T>`	A templated ‘smart pointer’ type, which `GCNode` objects use to refer to other `GCNode`s. Automatically adjusts the reference count of the node referred to. (In approaches using generational mark-sweep, it encapsulates the write barrier.)
`GCManager`	Controls the threshold(s) at which mark-sweep garbage collection will take place. (In approaches using generational mark-sweep, it also controls how many generations are collected.)
`GCRoot<T>` and `GCStackRoot<T>`	Smart pointer types used to provide long- and short-term protection against garbage collection.

Protection Using `GCStackRoot`

Short-term protection of `GCNode` objects from garbage collection is best achieved using `GCStackRoot` (though `PROTECT()`, `UNPROTECT()` etc. are still available).

Here's an (artificial) example that inserts a new link into a list following a specified location:

```
void insertAfter(ConsCell* location, RObject* car,
                 RObject* tag = 0)
{
    GCStackRoot<PairList> tail(location->tail());
    PairList* node
     = GCNode::expose(new PairList(car, tail, tag));
    location->setTail(node);
}
```

Protection Using `GCStackRoot`

Short-term protection of `GCNode` objects from garbage collection is best achieved using `GCStackRoot` (though `PROTECT()`, `UNPROTECT()` etc. are still available).

Here's an (artificial) example that inserts a new link into a list following a specified location:

```
void insertAfter(ConsCell* location, RObject* car,
                 RObject* tag = 0)
{
    GCStackRoot<PairList> ta  l(location->tail());
    PairList* nod
     = GCNode::ex              st(car, tail, tag));
    location->setTail(node);
}
```

The default is for the newly inserted node to have no tag: in CXXR `R_NilValue` is simply a null pointer.

Protection Using `GCStackRoot`

Short-term protection of `GCNode` objects from garbage collection is best achieved using `GCStackRoot` (though `PROTECT()`, `UNPROTECT()` etc. are still available).

Here's an (artificial) example that inserts a new link into a list following a specified location:

```
void insertAfter(ConsCell* location, RObject* car,
                 RObject* tag = 0)
{
    GCStackRoot<PairList> tail(location->tail());
    PairList* node
     = GCNode::expose(new PairList(car, tail, tag));
    locat               e);
}
```

`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.

Protection Using `GCStackRoot`

Short-term protection of `GCNode` objects from garbage collection is best achieved using `GCStackRoot` (though `PROTECT()`, `UNPROTECT()` etc. are still available).

Here's an (artificial) example that inserts a new link into a list following a specified location:

```
void insertAfter(ConsCell* location, RObject* car,
                 RObject* tag = 0)
{
    GCStackRoot<PairList> tail(location->tail());
    PairList* node
     = GCNode::expose(new PairList(car, tail, tag));
    location->setTail(node);
}
```

Create the new link. May carry out garbage collection.

Protection Using `GCStackRoot`

Short-term protection of `GCNode` objects from garbage collection is best achieved using `GCStackRoot` (though `PROTECT()`, `UNPROTECT()` etc. are still available).

Here's an (artificial) example that inserts a new link into a list following a specified location:

```
void insertAfter(ConsCell* location, RObject* car,
                 RObject* tag = 0)
{
    GCStackRoot<PairList> tail(location->tail());
    PairList* node
     = GCNode::expose(new PairList(car, tail, tag));
    location->setTail(node);
}
```

Expose the new link to garbage collection.

Protection Using `GCStackRoot`

Short-term protection of `GCNode` objects from garbage collection is best achieved using `GCStackRoot` (though `PROTECT()`, `UNPROTECT()` etc. are still available).

Here's an (artificial) example that inserts a new link into a list following a specified location:

```
void insertAfter(ConsCell* location, RObject* car,
                 RObject* tag = 0)
{
    GCStackRoot<PairList> tail(location->tail());
    PairList* node
     = GCNode::expose(new PairList(car, tail, tag));
    location->setTail(node);
}
```

The `GCStackRoot` object goes out of scope here, and its destructor automatically ends the GC protection it offers. No need to balance `PROTECT()`/`UNPROTECT()` 'by hand'.

Protection Using `GCStackRoot`

Short-term protection of `GCNode` objects from garbage collection is best achieved using `GCStackRoot` (though `PROTECT()`, `UNPROTECT()` etc. are still available).

Here's an (artificial) example that inserts a new link into a list following a specified location:

```
void insertAfter(ConsCell* location, RObject* car,
                 RObject* tag = 0)
{
    GCStackRoot<PairList> tail(location->tail());
    PairList* node
     = GCNode::expose(new PairList(car, tail, tag));
    location->setTail(node);
}
```

But GC-protection of '`tail`' isn't actually necessary here (assuming '`location`' is protected).

Outline

1. CXXR
2. The `RObject` Class Hierarchy
3. Memory Allocation and Garbage Collection
4. **Object Duplication**
5. Environments
6. Performance
7. Looking Forward

Duplicating `RObject`s

In CXXR, each `RObject` class defines whether and how objects of that class can be duplicated.

- Class `RObject` defines a method `clone()`, which by default returns a null pointer, indicating that the object is not clonable.
- Many classes in the `RObject` hierarchy define a copy constructor, which encapsulates the behaviour of CR's `Rf_duplicate` with respect to that class.
- Such classes usually also reimplement `clone()`, so that `foo->clone()` returns a pointer to a copy of `foo` made with the copy constructor.

Duplicating `RObject`s

In CXXR, each `RObject` class defines whether and how objects of that class can be duplicated.

- Class `RObject` defines a method `clone()`, which by default returns a null pointer, indicating that the object is not clonable.
- Many classes in the `RObject` hierarchy define a copy constructor, which encapsulates the behaviour of CR's `Rf_duplicate` with respect to that class.
- Such classes usually also reimplement `clone()`, so that `foo->clone()` returns a pointer to a copy of `foo` made with the copy constructor.

Duplicating `RObject`s

In CXXR, each `RObject` class defines whether and how objects of that class can be duplicated.

- Class `RObject` defines a method `clone()`, which by default returns a null pointer, indicating that the object is not clonable.
- Many classes in the `RObject` hierarchy define a copy constructor, which encapsulates the behaviour of CR's `Rf_duplicate` with respect to that class.
- Such classes usually also reimplement `clone()`, so that `foo->clone()` returns a pointer to a copy of `foo` made with the copy constructor.

`RObject::Handle<T>`

`RObject::Handle<T>` is yet another smart pointer template, which inherits from `GCEdge<T>`. Each `RObject::Handle<T>` encapsulates a pointer—possibly null—to a `T` object (where `T` is a type inheriting from `RObject`).

When a handle is copied it tries to copy the object it points to, by invoking `clone()`.

- If `clone()` succeeds (i.e. returns a non-null pointer), then the copied handle points to the copied `RObject`.
- If `clone()` returns a null pointer, then the copied handle points to the original `RObject`.

Use of `RObject::Handle` greatly simplifies the implementation of copy constructors in the `RObject` hierarchy: in some cases the default copy constructor supplied by the compiler does all that is required.

`RObject::Handle<T>`

`RObject::Handle<T>` is yet another smart pointer template, which inherits from `GCEdge<T>`. Each `RObject::Handle<T>` encapsulates a pointer—possibly null—to a `T` object (where `T` is a type inheriting from `RObject`).

When a handle is copied it tries to copy the object it points to, by invoking `clone()`.

- If `clone()` succeeds (i.e. returns a non-null pointer), then the copied handle points to the copied `RObject`.
- If `clone()` returns a null pointer, then the copied handle points to the original `RObject`.

Use of `RObject::Handle` greatly simplifies the implementation of copy constructors in the `RObject` hierarchy: in some cases the default copy constructor supplied by the compiler does all that is required.

`RObject::Handle<T>`

`RObject::Handle<T>` is yet another smart pointer template, which inherits from `GCEdge<T>`. Each `RObject::Handle<T>` encapsulates a pointer—possibly null—to a `T` object (where `T` is a type inheriting from `RObject`).

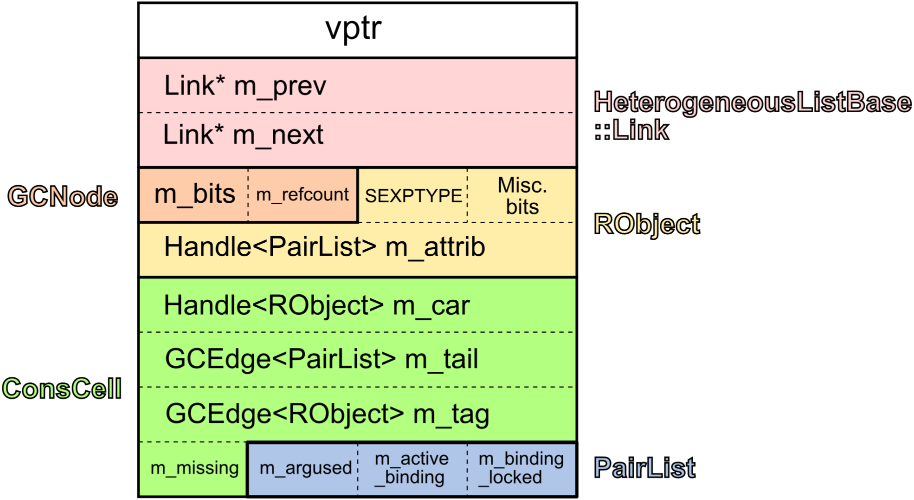
When a handle is copied it tries to copy the object it points to, by invoking `clone()`.

- If `clone()` succeeds (i.e. returns a non-null pointer), then the copied handle points to the copied `RObject`.
- If `clone()` returns a null pointer, then the copied handle points to the original `RObject`.

Use of `RObject::Handle` greatly simplifies the implementation of copy constructors in the `RObject` hierarchy: in some cases the default copy constructor supplied by the compiler does all that is required.

Layout of a `PairList` (`LISTSXP`) Object

(Schematic, for 32-bit architecture)



vptr

Link* m_prev

Link* m_next

HeterogeneousListBase ::Link

GCNode

m_bits

m_refcount

SEXPTYPE

Misc. bits

Handle<PairList> m_attrib

RObject

Handle<RObject> m_car

GCEdge<PairList> m_tail

GCEdge<RObject> m_tag

ConsCell

m_missing

m_argused

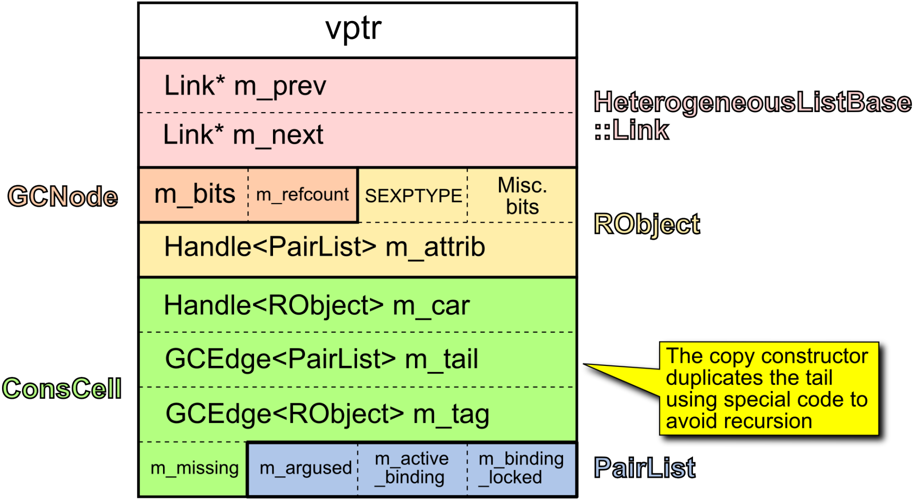
m_active _binding

m_binding _locked

PairList

Layout of a `PairList` (`LISTSXP`) Object

(Schematic, for 32-bit architecture)



vptr

Link* m_prev

Link* m_next

HeterogeneousListBase ::Link

GCNode

m_bits

m_refcount

SEXPTYPE

Misc. bits

Handle<PairList> m_attrib

RObject

Handle<RObject> m_car

GCEdge<PairList> m_tail

GCEdge<RObject> m_tag

ConsCell

m_missing

m_argused

m_active _binding

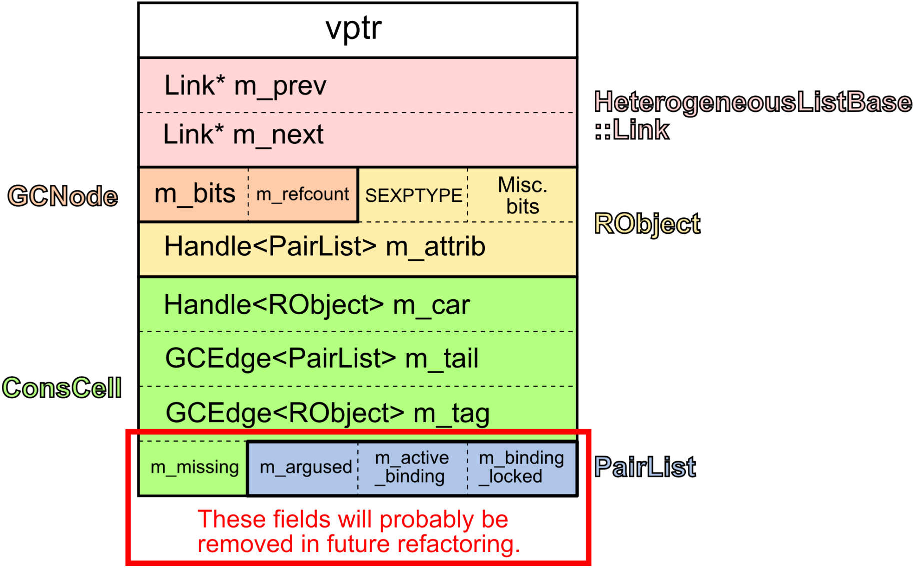
m_binding _locked

PairList

The copy constructor duplicates the tail using special code to avoid recursion

Layout of a `PairList` (`LISTSXP`) Object

(Schematic, for 32-bit architecture)



These fields will probably be removed in future refactoring.

Outline

1. CXXR
2. The `RObject` Class Hierarchy
3. Memory Allocation and Garbage Collection
4. Object Duplication
5. **Environments**
6. Performance
7. Looking Forward

Environments
General

- In CXXR, all environments are are implemented in the same way, including the base environment and the base namespace.
- Each `Environment` object comprises a pointer to its enclosing environment (null in the case of the empty environment) and a pointer to a `Frame` object.
- The base environment and the base namespace point to the same `Frame` (but have different enclosing environments).
- `Frame` is an abstract class defining a mapping from `Symbol`s (strictly, symbol addresses) to `Frame::Binding` objects, which in turn point to `RObject`s. (`Frame` inherits directly from `GCNode`, not from `RObject`.)

Environments
General

- In CXXR, all environments are are implemented in the same way, including the base environment and the base namespace.
- Each `Environment` object comprises a pointer to its enclosing environment (null in the case of the empty environment) and a pointer to a `Frame` object.
- The base environment and the base namespace point to the same `Frame` (but have different enclosing environments).
- `Frame` is an abstract class defining a mapping from `Symbol`s (strictly, symbol addresses) to `Frame::Binding` objects, which in turn point to `RObject`s. (`Frame` inherits directly from `GCNode`, not from `RObject`.)

Environments

General

- In CXXR, all environments are are implemented in the same way, including the base environment and the base namespace.
- Each `Environment` object comprises a pointer to its enclosing environment (null in the case of the empty environment) and a pointer to a `Frame` object.
- The base environment and the base namespace point to the same `Frame` (but have different enclosing environments).
- `Frame` is an abstract class defining a mapping from `Symbol`s (strictly, symbol addresses) to `Frame::Binding` objects, which in turn point to `RObject`s. (`Frame` inherits directly from `GCNode`, not from `RObject`.)

- In CXXR, all environments are are implemented in the same way, including the base environment and the base namespace.
- Each `Environment` object comprises a pointer to its enclosing environment (null in the case of the empty environment) and a pointer to a `Frame` object.
- The base environment and the base namespace point to the same `Frame` (but have different enclosing environments).
- `Frame` is an abstract class defining a mapping from `Symbol`s (strictly, symbol addresses) to `Frame::Binding` objects, which in turn point to `RObject`s. (`Frame` inherits directly from `GCNode`, not from `RObject`.)

Environments

Plus points

- Various forms of lazy loading can be encapsulated in classes inheriting from `Frame`.
- Likewise special `Frame` classes can provide functionality similar to the `RObjectTables` package, e.g. a frame that looks symbols up in a database.
- Hooks are provided to monitor when a `Frame`'s bindings are created, read or modified. These are used in provenance tracking.

Environments

Plus points

- Various forms of lazy loading can be encapsulated in classes inheriting from `Frame`.
- Likewise special `Frame` classes can provide functionality similar to the `RObjectTables` package, e.g. a frame that looks symbols up in a database.
- Hooks are provided to monitor when a `Frame`'s bindings are created, read or modified. These are used in provenance tracking.

Environments

Plus points

- Various forms of lazy loading can be encapsulated in classes inheriting from `Frame`.
- Likewise special `Frame` classes can provide functionality similar to the `RObjectTables` package, e.g. a frame that looks symbols up in a database.
- Hooks are provided to monitor when a `Frame`'s bindings are created, read or modified. These are used in provenance tracking.

Environments

Current limitations

- There is currently no analogue to CR's 'global cache'.
- All `Frame`s are currently implemented using class `StdFrame`, which uses the library class `std::tr1::unordered_map` (i.e. a hash table) to provide the symbol→binding mapping. For local environments, this probably imposes an excessive construction/destruction overhead.

Outline

1. CXXR
2. The `RObject` Class Hierarchy
3. Memory Allocation and Garbage Collection
4. Object Duplication
5. Environments
6. **Performance**
7. Looking Forward

Benchmarks

The following tests were carried out on a 2.8 GHz Pentium 4 with 1 MB L2 cache, comparing R-2.8.1 with CXXR revision 599, using comparable optimisation options.

Benchmark	**CR (secs)**	**CXXR trunk (secs)**	**CXXR gclab (secs)**
`bench.R` (Jan de Leeuw)	111	113	112
`kaltime10.R`	95	144	113
`stats-Ex.R`	30	61	69

A Benchmark Program: `kaltime10.R`

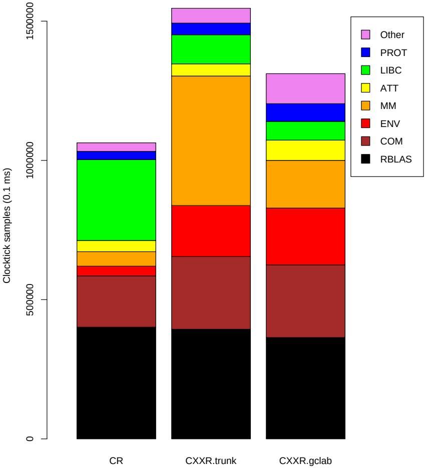
(Inspired by a Kalman filter time update.)

```
kaltime <- function(d, n){
    phi <- matrix(0.9/d, nrow=d, ncol=d)
    p <- matrix(0, nrow=d, ncol=d)
    q <- diag(d)
    for (i in 1:n) p <- phi %*% p %*% phi + q
    p
}

kaltime(10, 5000000)
```

Timing Comparison

`kaltime10.R`, 10×10 matrices, 5 000 000 iterations. Based on flat profile data from Intel VTune.



`Other` Anything not clearly attributable to the categories below.

`PROT` Initiating and ending protection from garbage collection.

`LIBC` `libc-2.9.so`

`ATT` Attributes.

`MM` Memory allocation, garbage collection, C++ destructors.

`ENV` Environments, inc. symbol look-up.

`COM` Common, i.e. functions little changed between CR and CXXR.

`RBLAS` `libRblas.so`: Basic linear algebra system.

Cache Performance

according to `cachegrind`

`kaltime10_100k.R`, 10×10 matrices, 100 000 iterations.

	CR	**CXXR (trunk)**	**CXXR (gclab)**
D1 misses	12 864 395 (0.4%)	36 063 852 (1.1%)	21 130 898 (0.6%)
L2d misses	6 655 104 (0.2%)	14 193 172 (0.4%)	373 466 (0.0%)
I1 misses	25 843 389 (0.4%)	31 854 095 (0.5%)	73 659 529 (1.2%)
L2i misses	63 323 (0.0%)	97 394 (0.0%)	55 400 (0.0%)
Brch mispred.	37 427 870 (5.4%)	41 119 592 (5.8%)	47 800 867 (6.3%)

Outline

1. CXXR
2. The `RObject` Class Hierarchy
3. Memory Allocation and Garbage Collection
4. Object Duplication
5. Environments
6. Performance
7. **Looking Forward**

Next Stages

- Upgrade to shadow R 2.9.1
- Factor out evaluation code (`Rf_eval()`) into the class hierarchy, and in the process reduce internal use of `PairList` objects in favour of lighter-weight data structures.
- Factor out serialization/deserialization code into the class hierarchy, and possibly switch to an XML serialization format.
- Refactorise R contexts (`RCNTXT`) and error handling in the spirit of C++ exception handling.
- Reintroduce a global cache for symbol lookup.

Header Files in `src/include/CXXR`

The interface to the core of CXXR is defined in header files in `src/include/CXXR`.

These are of two types:

`*.hpp` For use by C++ source files only.

`*.h` For use by C and C++ source files. These headers typically have the structure shown on the right.

```
#ifdef __cplusplus

Class definitions, etc.

extern "C" {
#endif  /* __cplusplus */

C-callable function definitions

#ifdef __cplusplus
}  // extern "C"
#endif
```

A C-Callable Function

C++ sees inlining; C doesn't

In `RealVector.h`:

```
#ifndef __cplusplus
    double* REAL(SEXP x);
#else
    inline double* REAL(SEXP x)
    {
        using namespace CXXR;
        return &(*SEXP_downcast<RealVector*>(x))[0];
    }
#endif
```

In `RealVector.cpp`

```
namespace CXXR {
    namespace ForceNonInline {
        double* (*REALp)(SEXP) = REAL;
    }
}
```

This forces the C++ compiler to generate a non-inlined embodiment of the function `REAL()`, which is what C source files will link to.

`GCNode` Lists

The live list

Live list

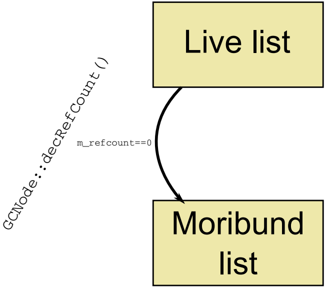
Moribund list

Class `GCNode` arranges all `GCNode` objects on a number of lists, implemented as static class members. Principal among these are the **live list**, `s_live`, and the **moribund list**, `s_moribund`.

Newly created `GCNode` objects are placed on the live list.

`GCNode` Lists

The moribund list



When a `GCNode`'s reference count falls to zero (having previously been non-zero), it is transferred to the moribund list.

It is quite common for the reference count subsequently to rise back above zero, so these nodes cannot be immediately deleted.

`GCNode` Lists

`GCNode::operator new`

Live list

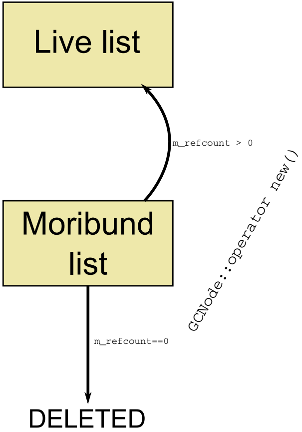
Moribund list

When an object of a type derived from `GCNode` is created, memory for it is automatically obtained using the function `GCNode::operator new()` (rather than the general-purpose `::operator new()`).

This function carries out garbage collection as explained next.

GCNode Lists

`GCNode::operator new`



Live list

`m_refcount > 0`

Moribund list

`m_refcount==0`

DELETED

`GCNode::operator new()`

Operation of `GCNode::operator new()`:

1. Scan through the moribund list. Any node whose reference count is still zero is deleted; other nodes are restored to the live list.
2. If, after Step 1, `MemoryBank::bytesAllocated()` exceeds a threshold value, initiate a mark-sweep garbage collection.
3. Finally, allocate the requested memory by calling `MemoryBank::allocate()`.

Protection Using `GCRoot`

`GCStackRoot` objects must be destroyed in the reverse order of their creation, and are best suited to stack-based (automatic) variables.

Longer-term protection from garbage-collection is better achieved using the `GCRoot<T>` smart pointer template, which works in the same way, but is not subject to this restriction. For example entities such as `R_BlankString` are permanently protected by `GCRoot`s in CXXR.

However, construction and destruction of `GCRoot`s is more time-consuming than for `GCStackRoot`.

Protection Using `GCRoot`

`GCStackRoot` objects must be destroyed in the reverse order of their creation, and are best suited to stack-based (automatic) variables.

Longer-term protection from garbage-collection is better achieved using the `GCRoot<T>` smart pointer template, which works in the same way, but is not subject to this restriction. For example entities such as `R_BlankString` are permanently protected by `GCRoot`s in CXXR.

However, construction and destruction of `GCRoot`s is more time-consuming than for `GCStackRoot`.

A GC Problem in C++

Suppose that `Foo` is a class that inherits from `GCNode`. Here's how creating a new `Foo` object would naturally be written in C++:

```
Foo* f = new Foo(...);
```

Or perhaps:

```
GCStackRoot<Foo> f
  = new Foo(...);
```

The `new Foo(···)` expression gets compiled into this:

1. `GCNode::operator new` is called to allocate sufficient memory to hold a `Foo` object; may initiate mark-sweep.
2. A constructor of `GCNode` is called to initialise the `GCNode` part of the object;
3. Then `Foo`'s constructor initialises the `Foo` part of the object;
4. Finally, a pointer to the constructed object is returned and assigned to `f`.

A GC Problem in C++

Suppose that `Foo` is a class that inherits from `GCNode`. Here's how creating a new `Foo` object would naturally be written in C++:

```
Foo* f = new Foo(...);
```

Or perhaps:

```
GCStackRoot<Foo> f
  = new Foo(...);
```

The `new Foo(···)` expression gets compiled into this:

1. `GCNode::operator new` is called to allocate sufficient memory to hold a `Foo` object; may initiate mark-sweep.
2. A constructor of `GCNode` is called to initialise the `GCNode` part of the object;
3. Then `Foo`'s constructor initialises the `Foo` part of the object;
4. Finally, a pointer to the constructed object is returned and assigned to `f`.

A GC Problem in C++

Suppose that `Foo` is a class that inherits from `GCNode`. Here's how creating a new `Foo` object would naturally be written in C++:

```
Foo* f = new Foo(...);
```

Or perhaps:

```
GCStackRoot<Foo> f
  = new Foo(...);
```

The `new Foo(···)` expression gets compiled into this:

1. `GCNode::operator new` is called to allocate sufficient memory to hold a `Foo` object; may initiate mark-sweep.
2. A constructor of `GCNode` is called to initialise the `GCNode` part of the object;
3. Then `Foo`'s constructor initialises the `Foo` part of the object;
4. Finally, a pointer to the constructed object is returned and assigned to `f`.

What happens if `Foo`'s constructor needs to create a subobject—also a `GCNode`—and that results in a mark-sweep garbage collection?

Solution: Infant Immunity

A solution to the problem is for each `GCNode` to be *immune* from garbage collection while it is under construction. The completion of construction is signalled using this idiom:

```
Foo* f = GCNode::expose(new Foo(...));
```

(`GCNode::expose()` here is an identity function with the side effect of exposing its argument to garbage collection.)

Implementations of Infant Immunity

1. Have the sweep phase ignore infant nodes.
   Snag: Exposure needs to recurse to subobjects.
2. Regard infant nodes as reachable.
   Snag: The mark phase may visit nodes that are still under construction, and contain junk pointers.
3. Inhibit mark-sweep GC entirely while any `GCNode` is under construction.
   The solution currently favoured.

Implementations of Infant Immunity

1. Have the sweep phase ignore infant nodes.
   Snag: Exposure needs to recurse to subobjects.
2. Regard infant nodes as reachable.
   Snag: The mark phase may visit nodes that are still under construction, and contain junk pointers.
3. Inhibit mark-sweep GC entirely while any `GCNode` is under construction.
   The solution currently favoured.

Implementations of Infant Immunity

1. Have the sweep phase ignore infant nodes.
   Snag: Exposure needs to recurse to subobjects.
2. Regard infant nodes as reachable.
   Snag: The mark phase may visit nodes that are still under construction, and contain junk pointers.
3. Inhibit mark-sweep GC entirely while any `GCNode` is under construction.
   The solution currently favoured.