R documentation

of all in '../../man'

August 12, 2009

R topics documented:

`as.field.decl` *Find appropriate sub-node*

Description

These functions attempt to search the given node in a sensible manner to find the relevant sub-node as indicated by the name of the function. Fror example, `as.parm.decl` will search a `GCC::Node::function_decl` and find the first `GCC::Node::parm_decl` node, if it exists.

These are meant as convenience functions to find map from a high-level object to the first node of the target type so that we can then iterate over the nodes, starting at that one.

Usage

```
as.field.decl(x, follow = TRUE, acceptableTypes =
                 c("GCC::Node::field_decl", "GCC::Node::var_decl",
                 "GCC::Node::const_decl"))
as.record.type(x, stopOnFail = TRUE)
as.parm.decl(x)
as.type.decl(x)
```

Arguments

`x`	a reference to a Perl GCC::Node object obtained from parsing the translation unit.
`stopOnFail`	a logical value indicating whether to generate an error if the node could not be mapped to a record_type or simply return `NULL`
`follow`	(logical) controls whether to jump from the given node to one of its elements to find the definition of the field.
`acceptableTypes`	a character vector giving the names of the classes that are legitimate nodes for a field.

Value

A reference to a Perl GCC::Node object or an error.

Author(s)

Duncan Temple Lang <duncan@wald.ucdavis.edu>

References

http://www.omegahat.org/RGccTranslationUnit

See Also

parseTU

---

boolType-class *Classes for representing primitive/built-in data types.*

Description

These are classes for representing data types in C code for the built-in data types in the C language, e.g. int, double, float, etc. At present, we have the basic classes and differentiate between, e.g. int and unsigned int, based on the name in the class. This is typically detected from the name in the translation unit.

These classes are intended to be instantiated as part of resolving type information in the translation units via resolveType.

Objects from the Class

Typically instances of these classes are created in the evaluation of a call to resolveType. However, objects can be created directly by calls of the form new("boolType", ...). The resolveType tends to get the name right from the information in the translation unit nodes.

Slots

name: Object of class "character". This gives the human readable name such as "int", "char", "unsigned int", etc.

Extends

Class "BuiltinPrimitiveType", directly. Class "TypeDefinition", by class "BuiltinPrimitiveType

Methods

No methods defined with class "boolType" in the signature.

Author(s)

Duncan Temple Lang <duncan@wald.ucdavis.edu>

References

http://www.omegahat.org/RGccTranslationUnit

`computeGlobalConstants` *Find the top-level/global variables that are constant*

Description

This function searches the translation unit for non-local or top-level C/C++ variables and determines whether they are constant (i.e. declared with the qualifier `const`). The function then generates C code which can be run and in turn outputs R code to define R variables with the same values as these C level constants. This works for simple types.

Usage

```
computeGlobalConstants(tu = NULL, gvars = getGlobalVariables(tu, files),
                       files = character(),
                        defs = DefinitionContainer(tu),
                          varsOnly = FALSE, symbolic = FALSE,
                            access = c("public", "protected"))
```

Arguments

`tu`	the translation unit node/parser, obtained by a call to `parseTU`
`gvars`	the global variable nodes. In some cases, these will already have been computed and then one only needs to pass these and not `files`.
`files`	a character vector identifying the names of the files mentioned in the TU nodes in which we are interested, i.e. those in which the variables of interest are declared/defined. If `gvars` is specified, this is not used.
`defs`	the table of resolved nodes and data types and routines. This is a mutable object of class `DefinitionContainer` and we typically have just one of these for each translation unit and we reuse it across many of these functions which process the TU nodes further and resolve the different types.
`varsOnly`	a logical value which, if `TRUE` merely returns the indices of the elements in `gvars` which are constant. In general, the caller will want the generated code to calculate these values and so the default of `FALSE` is appropriate.
`symbolic`	a logical value indicating whether we want the values of the constants to be returned as values or symbolically in terms of other constant variables, e.g. `int x = a + b;` as `7` if `a = 3` and `b = 4` or as the expression `a + b`
`access`	a character vector containing any of "public", "protected" and "private" which is used to filter the constants returned. This is relevant only for C++ code. This specifies whether we want public, protected and/or private global variables.

Value

An object of S3 class `ComputeConstants`. This is a list with two elements:

`cmds`	a character vector with names giving the names of the global variables and entries giving C/C++ code to calculate the constant value or an `NA`

`filenames` a character vector that parallels cmds and gives the names of the header file in which the variable is declared. This is used when we emit the C code to compute the values of these constants as R values as we need to include the header files to ensure the constants are defined and accessible to our generated code. As with all TU files, we don't have the full path name but merely the base name of the file and some more computations are warranted to get the proper access to these files for the `#include` in the generated code.

Author(s)

Duncan Temple Lang

References

The GCC compiler suite

See Also

`parseTU` `computeGlobalEnumConstants` `getCppDefines`

Examples

```
tu = parseTU(system.file("examples", "globals", "globals.c.tu", package = "RGCCTranslat
g = computeGlobalConstants(tu)
names(g)
names(g$cmds)
g$filenames
```

`computeGlobalEnumConstants`

Compute top-level enumeration definitions

Description

This function processes the translation unit nodes and finds the enumeration definitions, i.e. the symbolic constants that are defined at the C/C++ level using the `enum` keyword. The result are named values of numerical values. In some cases, the enumeration collection has a name by way of a typedef. Additionally, some enumeration collections are simple "sequence" of values (some times not contiguous) which is used primarily to differentiate the values. It acts like a factor. In other cases, the values are intended to be OR'ed together in C to indicate one or more combined states. These are bitwise values. This function attempts to guess which type an enumeration definition falls into and returns the information in the appropriate class, i.e. `EnumerationDefinition` or `BitwiseEnumerationDefinition`.

The function then returns the collection of both named and anonymous enumeration collections which can be used to create corresponding variables for use in R.

Usage

```
computeGlobalEnumConstants(tu = NULL, enums = getEnumerations(tu, files),
                           files = character(),
                            defs = DefinitionContainer(tu), anonymousOnly = FAL
```

Arguments

tu	the translation unit nodes obtained from a call to parseTU
enums	the list of TU nodes that define the enumeration definitions of interest.
files	a character vector giving the names of the files in which the enumeration definitions of interest are located. This is used when calculating the default value of enums and is used to eliminate enumeration definitions for files such as system header files for which we do not want the definitions.
defs	a DefinitionContainer that is used for resolving the values of the different types and nodes with the TU.
anonymousOnly	a logical value indicating whether we only want the anonymous, i.e. not typedef'ed enumeration collection definitions.

Value

An object of class TopLevelEnumDefs. This is a list in which entry is an object of class EnumerationDefinition, with some being BitwiseEnumerationDefinition objects identifying bitwise enumerations.

Author(s)

Duncan Temple Lang

References

The GCC compiler suite

See Also

computeGlobalConstants

Examples

```
tu = parseTU(system.file("examples", "distance.c.t00.tu", package = "RGCCTranslationUnit
e = computeGlobalEnumConstants(tu)
table(sapply(e, class))
names(e)

# the anonymous enums
grep("^[.]", names(e), value = TRUE)

 # the named entries
grep("^[^.]", names(e), value = TRUE)
```

`convertRValue` *Create C/C++ code to convert R value to C/C++ type*

Description

These three functions (and associated methods) perform the code generation to marshal values between R and C/C++. `coerceRValue` is used in the R wrapper function to ensure that the argument is coerced to the appropriate type in R before being passed to the C code. This can perform explicit coercion to change the object and also raise errors if it is not appropriate. `convertRValue` generates C/C++ code to convert an R value given as a `SEXP` to the specified target type in C/C++. And `convertValueToR` is used for converting native types to their R equivalents when they are being returned to R, e.g. as return values of routines or copying fields in a struct/class back to R.

Usage

```
convertRValue(to, name, parm, parameters, typeMap = list(), helperInfo = NULL)
coerceRValue(name, parm, caller = NULL, typeMap = list(), helperInfo = NULL)
convertValueToR(name, parm, parameters, invoke = "", typeMap = list(),
                out = FALSE, helperInfo = NULL, ...)
```

Arguments

`to`	the description of the native type to which the R value is to be converted.
`name`	the name of the C-level variable holding the R object.
`parm`	the type description for the target parameter. This is an object derived from `TypeDefinition-class`
`parameters`	a named list of the other paramter types. (Is this sometimes a character vector with just the names?)
`typeMap`	an object that provides customized options for different C/C++ types for any of the three different operations.
`invoke`	?
`helperInfo`	
`out`	
`caller`	
`...`	

Value

These return code segments.

Author(s)

Duncan Temple Lang

createClassBindings *Generate R and C/C++ code bindings for a C++ class.*

Description

This function is used to generate code that provides an R-language interface to a C++ class. Currently, it generates code to access the instances of the class as references. This makes sense for C++ classes.

Usage

```
createClassBindings(def, nodes, className = rdef@name, # rdef computed in functi
                    types = DefinitionContainer(nodes),
                    polymorphicNames = unique(names(mm)[duplicated(names(mm))]),
                    abstract = isAbstractClass(mm, nodes),
                    resolvedMethods = resolveType(getClassMethods(def), nodes, t
                    typeMap = list(),
                    generateOverloaded = TRUE, ifdef = character(),
                    helperInfo = NULL, access = "public",
                    dynamicCast = list(),
                    otherClassMethods = NULL, useClassNameMethod = FALSE,
                    signatures = list(),
                    useSignatureParameters = TRUE,
                    dispatchInfo = data.frame(),
                    defaultBaseClass = if(useClassNameMethod)
                                          "RC++ReferenceUseName"
                                       else
                                          "RC++Reference",
                    classDefs = NULL, ...)
```

Arguments

def	the node giving the class definition. In the future, we will allow this object to be a fully resolved description of the class, i.e. with the methods and types resolved.
nodes	the TU parser returned from parseTU which is an array of the tu nodes.
className	the name of the C++ class being processed
types	the collection of resolved data types, routines, etc. typically a DefinitionContainer.
polymorphicNames	a character vector giving the names of the methods that are overloaded, typically just within this class.
abstract	a logical value indicating if this class is an abstract class for which there can be no C++-level instances.
resolvedMethods	a list of the fully resolved methods, typically obtained from a call to resolveType on the methods returned by getClassMethods. This can be either a list of all the resolved methods

`typeMap` a user-specifiable list of "hints" for mapping particular data types in the generated code, i.e. for converting between R and C/C++ and coercing in R functions to the appropriate type for the C/C++ method.

`...` additional arguments that are passed on to `createMethodBinding`.

`generateOverloaded`
a logical value XXX

`ifdef` a character string which, if non-empty, is used to enclose the entire generated code within and `#ifdef string ... #endif` block. This allows one to generate code that is conditionally compiled, e.g. for specific platforms.

`helperInfo` passed on to each call to `createMethodBinding`.

`access` a character vector containing one or more of public, protected, private to indicate which methods should be processed. For each method, we compare its accessability value to this vector and only process it if there is a corresponding entry in this vector. Thus, a value of `c("public", "protected")` will process both the public and protected methods.

`dynamicCast`

`otherClassMethods`

`useClassNameMethod`

`signatures`

`useSignatureParameters`

`dispatchInfo`

`defaultBaseClass`

`classDefs`

Note

For regular C/C++ structures, we can access the fields and work with instances rather than references to instances, passing such objects between R and C/C++ representations by explicitly copying the fields.

Author(s)

Duncan Temple Lang <duncan@wald.ucdavis.edu>

References

http://www.omegahat.org/RGCCTranslationUnit

See Also

`writeCode` `parseTU`

`createDerivedClass` *Generate code for a derived C++ class with R functions as methods*

Description

The goal of this function is to define a new C++ class that extends the specified C++ class and which allows implementations of the virtual methods within that class (and ancestors) via R functions. This allows us to override some or all of the methods entirely within R and have those R functions be transparently available to other C++ code. One can create an instance of this new C++ class from within R and specify functions for some of the methods either during the construction or during the life of that object. In this way, the class' methods become dynamic. This mechanism is thus very convenient for exploratory prototypying where we can update methods to handle new situations and fix bugs.

We can only override virtual methods in the C++ class, i.e. those methods that are declared virtual in that class or any of its ancestors.

Specifying an R function to use for a C++ method becomes slightly more troublesome when the method is overloaded, i.e. has multiple versions with different signatures.

This function attempts to deal with various forms of inputs and to do it somewhat efficiently by avoiding recomputing information from the translation unit.

If called with one or more class names and the TU nodes it resolves the class nodes and methods Essentially, if one wants to compute the class nodes ahead of time and resolve all their methods, this information can be reused and no computations duplicated uncessarily. It can work with resolved class definitions or the raw class nodes in the tu or with just the name of the class(es) of interest.

The class names of interest can be given as a character vector and the corresponding R names can be given as the names of this character vector. Any missing names are computed using the default naming scheme.

If one has computed the class nodes and resolved their methods already, then it is simplest to loop over the classes of interest and call this function for each of these. See the example below.

Usage

```
createDerivedClass(className, tu, types = DefinitionContainer(tu),
                   classNodes = getClassNodes(tu), methods = NULL,
                   classDefs = getClassDefs(tu, klasses = classNodes),
                   virtualOnly = TRUE, inheritanceStyle = "public",
                   typeMap = list(),
                   RClassName = paste("R", className, sep = ""),
                   RNativeBaseClassName = c(native = "RDerivedClass", r = "RDeri
                  signatures = list(), dispatchInfo = data.frame(),
                   ...)
```

Arguments

`className`	a value identifying the class which we want to extend, i.e. the base class of our soon-to-be-created new, derived class. This can be the name of the class given as a string or alternatively a node in the translation unit that identifies the class definition, e.g. returned from a call to `getClassNodes`. One can specify a character vector with more than one element. The elements identify different existing classes by name and the code for an extended class will be developed

for each of these base classes. (These are not treated as a collection of superclasses for a single new derived base class.) If the character vector has names, these are used as the names for the new, extended classes instead of the value of the `RClassName` argument.

`tu` the translation unit nodes obtained via a call to `parseTU`

`types` a DefinitionContainer object which is a mutable object (an environment) used to collect all the resolved TU nodes, data types, routines, etc. so as to avoid having to repeat the same calculations each time a node is needing to be resolved in a different context.

`classNodes` a list of the nodes in the translation unit object (`tu`) which identify the ancestor classes of the newly defined class. If `tu` is specified, the default argument is usually appropriate.

`methods` a list of the methods for all of the ancestor classes of the newly defined classes. If the `tu` object is provided, these can be computed automatically.

`virtualOnly` a logical value indicating whether to provide overriding methods only for the virtual methods defined in the base class or any of its ancestors.

`typeMap` an object of S3-style class `TypeMap` which provides information about the relationships between C/C++ types and corresponding R types, i.e. C to R type equivalencies, routines and R functions for converting from one to the other, etc.

`inheritanceStyle`
a string specifying the nature of the inheritance, i.e. one of "public" or "protected"

`RClassName` the name to use for the new C++ class being defined.

`...`

`classDefs`
`RNativeBaseClassName`

`signatures`

`dispatchInfo`

Value

An list with several components:

`classDefinition`
the C++ code defining and implementing this new class

`rsetClass` the R code defining the R version of this class.

`rsetMethods` the C++ routine that can be called from R to set the R functions to be used as methods for this class.

`setMethods` the C++ routine that actually sets these methods in the C++ instance/object

`rsetMethodsFunction`
the R function that sets the method functions within an instance of the newly created class, i.e. that calls the rsetMethods routine

`rfieldAccessors`
R method for accessing the fields in the base class(es) and ancestor classes

...

Author(s)

Duncan Temple Lang

See Also

getClassNodes resolveType DefinitionContainer

Examples

```
library(RGCCTranslationUnit)
  # the next 5 commands are something that we do generally
  # when processing code
tu = parseTU(system.file("examples", "shapes.cc.t00.tu", package = "RGCCTranslationUnit"))

z = createDerivedClass("Ellipse", tu)

# Two classes in one action and amortize the cost of calculating the
# class nodes and resolving methods.
z = createDerivedClass(c("Rectangle", "Ellipse"), tu)

# This version explicitly computes the class nodes and resolves the methods
# presumably for use in other circumstances, e.g. generating the bindings to
# the existing classes and

types = DefinitionContainer(nodes = tu)
classNodes = getClassNodes(tu, "shapes")

methods = lapply(classNodes, function(n) resolveType(getClassMethods(n), tu, types))
zz =
 lapply(c("Rectangle", "Ellipse"),
        function(k) {
         def = resolveType(classNodes[[k]], tu, types)
         createDerivedClass(def, methods = methods[c(k, def@ancestorClasses)])
        })

if(identical(zz, z))
  cat("Same results with both approaches\n")


#
names(classNodes)
createDerivedClass(c(RRectangle = "Rectangle", RCircle = "Circle", REllipse = "Ellipse"),



klasses = getClassNodes(tu, "shapes")
methods = lapply(klasses, getClassMethods)
def = DefinitionContainer(nodes = tu)

rect = resolveType(klasses[["Rectangle"]], tu, def)

resolvedMethods = lapply(methods, function(x) lapply(x, resolveType, tu, def))

  # This is the command that is specific to creating the code defining
  # a derived class.
code = createDerivedClass(rect, methods = resolvedMethods["Rectangle"],
```

```
                              RClassName = "R_Rectangle")

code = createDerivedClass(rect, "R_Rectangle", methods = resolvedMethods["Rectangle"], vi

#
ellipse = resolveType(klasses[["Ellipse"]], tu, def)
code = createDerivedClass(ellipse, methods = methods, virtualOnly = TRUE)

#"R_Ellipse", resolvedMethods[c("Ellipse", "Circle", "Shape")],
#                            "Ellipse", getFields(klasses[[c("Ellipse")]]),
#                            virtualOnly = TRUE)
```

`createInterface` *Top-level function for generating code to interface to code in a translation unit.*

Description

The idea is that this will be the top-level entry point which a caller can use to generate an interface to a translation unit. This would then generate all the code for interfacing to the code in that translation unit. The results could then be written via `writeCode` and the resulting C/C++ and R code would be runnable, assuming the availability of the dependencies such as RAutoGenRunTime.

Usage

```
createInterface(decls, files = character(0), ...)
```

Arguments

`decls`	the translation unit parser object returned from e.g. `parseTU` which is a vector of the nodes in the tu file.
`files`	the names of the files (without the directory part identifying the files in which the definitions of interest are located. This is used to filter our the symbols that are not of interest, e.g. the system level symbols.
`...`	

Author(s)

Duncan Temple Lang

References

The GCC suite and translation units

See Also

`parseTU` `createClassBindings`

createMethodBinding *Generate R and C/C++ code to interface to an C++ method*

Description

This function generates text defining an R function and a corresponding C/C++ routine that can be used to invoke the underlying method from R. The code takes care of marshalling the arguments from R to C/C++ and onto the original routine and converting the answer back to an R value.

In the future, it will be easier to customize this conversion.

Usage

```
createMethodBinding(m,
                    className = if (is(m, "ResolvedNativeClassMethod"))
                                   m$className
                                else
                                   character(),
                    isPolymorphic = FALSE,
                    addRFunctionNames = !isPolymorphic && length(inheritedClassN
                    name = m$name,
                    nodes = list(),
                    typeMap = list(),
                    paramStyle = if("paramStyle" %in% names(m))
                                    m$paramStyle
                                 else
                                    rep(NA, length(m$parameters)),
                    defaultRValues = list(),
                    ifdef = character(),
                    helperInfo = NULL,
                    resolvedMethods = list(),
                    extern = length(className) > 0,
                    useClassNameMethod = FALSE,
                    force = FALSE, signature = NULL,
                    dispatchInfo = data.frame(),
                    useSignatureParameters = TRUE)
```

Arguments

m	the description of the method, with the parameter and return type information fully resolved.
className	the name of the C++ class to which the method belongs.
isPolymorphic	a logical value
addRFunctionNames	a logical value indicating whether to assign R functions to a name. By default, if the method has multiple versions, i.e. is polymorphic, then the function is anonymous so that these functions can be used in other composite functions/methods.

name — the name of the C++ method being processed. This is usually available from the method itself, but can be specified here to override that default name or if, for some reason, it is not available. This allows us to specify the name for the wrapper routine to disambiguate between different overloaded methods. This essentially allows us to override the general action if `isPolymorphic` is `TRUE`.

If this character vector has 2 or more elements, the second element is taken as the name of the R function and the first element is the name of the C/C++ routine.

nodes — the array of TU nodes, typically returned from a call to `parseTU`. This is used to resolve other nodes within the TU graph if this is necessary.

typeMap — a list whose elements indicate how to map a value between R and C/C++. Each element should be a list containing 4 elements named target, coerceR, convertToC, convertToR. The target is an object that describes what to match. This should currently be an explicit `TypeDefinition-class` object of the appropriate sub-class. The other elements are eithers strings identifying R or C/C++ function or routine names that perform the relevant operation. The coerceR entry is used in the R wrapper function for a C/C++ routine when coercing the R argument to the approriate type before passing it to the C/C++ routine in the `.Call`. convertToC is used in the C/C++ wrapper routine to marshall the value from an R object (`SEXP`) to a C/C++ value. And convertToR is used when converting a native C/C++ value back to an R object.

The converter elements can also be functions rather than strings.

paramStyle — a character vector with the same length as the parameter list (even for a constructor) with elements which are simply empty strings, in, out or inout indicating whether the parameters are to be treated as standard (""), input only with no interest in the output ("in"), only for output and so not accepted as arguments in the call ("out"), and both input and output arguments. out parameters are collapsed in the bindings so that they do not appear in the signature of the R function, but are returned in the result. When a routine has any inout or out arguments, the result becomes a list containing the actual return value from the C/C++ routine and the out and inout values named according to the parameter names.

defaultRValues — a named character vector giving default R expressions/values for the parameters in the R function. One need not specify values for all parameters, only those of interest. If no value is supplied for a parameter, any value in the parameter's `defaultValue` field is used, so one can set this prior to the call in the `m` object. The translation unit parser will find default values for C++ routines and methods when they are identified in the code and will make an attempt to create the corresponding R expression.

This parameter will be generalized to accept a list of expressions/calls or strings.

ifdef — a character string which, if non-empty, is used to enclose the entire generated code within and `#ifdef string ... #endif` block. This allows one to generate code that is conditionally compiled, e.g. for specific platforms.

resolvedMethods

helperInfo

extern

useClassNameMethod

force

`signature`

`dispatchInfo`

`useSignatureParameters`

Value

A list containing the generated code text for each of the languages, native (C/C++) and R.

`native`	text for the C/C++ routine
`r`	text for the R function (with no variable name/assignment information)

Note

In the future, we could allow for name to be a vector of length 2 and this would give us the name of the C routine and the name of the C++ method.

Author(s)

Duncan Temple Lang <duncan@wald.ucdavis.edu>

References

http://www.omegahat.org/RGCCTranslationUnit

See Also

`readTU` `getClassNodes` `getClassMethods` `resolveType`

Examples

```
 library(RGCCTranslationUnit)
 my = parseTU(system.file("examples", "ABC.cpp.tu", package = "RGCCTranslationUnit"))

 k = getClassNodes(my)
 m = getClassMethods( k$B )
 s = resolveType(m$shift, my)

 z = createMethodBinding(s, "B")

#XXX
 p = parseTU(system.file("examples", "structs.cpp.t00.tu", package = "RGCCTranslationUnit
 d = getAllDeclarations(p, "structs")
 r = getRoutines(p, "structs")

 # RGCCTranslationUnit:::processFunction( p [[ r[[ "createA" ]][["INDEX"]]  ]])
```

createProxyRCall *Generate C routine that calls R function*

Description

This function is used to create a C routine that calls an R function, passing the R function the values it received when it was called. In this way, the C routine acts as a wrapper/proxy for an R function. This is of use when we wish to use an R function when a C routine is expected, i.e. as a C function pointer. The generated C routine merely marshals the inputs to the R function and the output from the R function to the caller of the C routine.

This code generation mechanism allows the generator to specify how the R function to be called is computed at run-time. (See `functionVar`.)

This same approach is used to create proxy functions in C++ that act as methods for a class and which call R functions to implement the method.

Usage

```
createProxyRCall(func, name, functionVar = paste("R", name, "function_var", sep
```

Arguments

`func`	a description of the function pointer for which we are to generate this proxy. This should be an object of class `FunctionPointer`.
`name`	the name to give the C routine.
`functionVar`	a string giving C code that is executed to get the R function object which is to be called. This can be the name of a global variable, or an accessor into one of the parameters of the C routine to fetch application-specific data and cast it to an R function type in C (SEXP).
`preamble`	a character vector giving additional code that is to be inserted into the body of the generated routine. This will appear just after the local declarations of this routine. This code can therefor be further declarations, e.g. to create local variables for storing intermediate computations, and/or tests to verify the function object is valid, or indeed any of the parameters.

Value

An object of class CRoutineDefinition

Author(s)

Duncan Temple Lang

See Also

`parseTU` `resolveType` `generateInterface`

`CRoutineDefinition` *Create a code object*

Description

These constructor functions are used to create a text representation of code that we can output later. We can represent a C routine or an R function with the actual definition and also its name and declaration/prototype for C routines.

Usage

```
CRoutineDefinition(name, code, nargs = NA,
                   declaration = getDeclaration(code),
                   className = character(),
                   obj =  new("CRoutineDefinition"),
                   formatCode = TRUE)
```

Arguments

`name`

`code`

`obj`

`nargs`

`className`

`declaration`

`formatCode`

Value

An object of class `CRoutineDefinition-class` or `RFunctionDefinition-class` depending on which constructor is invoked!

Note

The RCode function uses S3-style classes and is less structured than the C routine representation as it does not need to know the name of the function, etc. but is merely a textual representation of R code.

Author(s)

Duncan Temple Lang

See Also

Any functions that we use to generate a C/C++ routine definition or an R function or R code.

CRoutineDefinition-class
Class "CRoutineDefinition"

Description

This hierarchy of classes from CodeDefinition to RFunctionDefinition and CRoutineDefinition are used as a simple representation of text that is code for a particular function/routine entity.

Objects from the Class

Use the constructor functions `CRoutineDefinition` and `RFunctionDefinition` to create an instance of the appropriate class. These take care of fixing the code to indent "appropriately" and put it into a single string. See the internal function `formatCode` for how this is done if you want to enhance it.

Slots

`code`: Object of class `"character"` giving the name of the routine, i.e. how one would call it and not the declaration.

`name`: Object of class `"character"` the entire code for the routine in a single string

`prototype`: Object of class `"character"` that provides the prototype or declaration for the routine that would be put in a header/include file.

Methods

No methods defined with class "CRoutineDefinition" in the signature.

Author(s)

Duncan Temple Lang

See Also

The internal function `formatCode` can be used to format code.

Examples

```
CRoutineDefinition("square", c("double",
                               "square(double x)",
                               "{",
                               "return( x  * x);",
                               "}"))
```

`DefinitionContainer` *Create a DefinitionContainer object for managing resolved type nodes*

Description

This function is the constructor function for creating a DefinitionContainer object. Such an object is simply an environment that we use for mutability across function calls as we recursively process nodes and across top-level calls to `resolveType`. This manages the types returned by `resolveType` and avoids reprocessing the same node from different paths in the graph. Also, this therefore avoids problems of resolving recursively defined nodes, e.g. structs with an element which is a pointer to that type being defined such as a linked list.

There are names and apply methods for finding out what is in the container and looping over the elements.

Usage

```
DefinitionContainer(parser = NULL, e = structure(new.env(TRUE), class = "Definit
                    nodes = NULL, reportDuplicates = FALSE,  verbose = FALSE, f
```

Arguments

`parser`	the parser, i.e. array of nodes to be associated with the node-type mapping. Don't use a definition container populated from one set of nodes with a different set of nodes!
`e`	the environment to use
`nodes`	
`reportDuplicates`	a logical value which specifies whether to emit a warning if two different nodes are resolved and map to the same human-readable name, i.e. other than their node index name.
`verbose`	a logical value which if `TRUE` causes lots of information to be displayed on the console when the container is managing the transactions of querying and inserting the table.
`force`	a logical value indicating whether to force the creation of a new `DefinitionContainer` (`TRUE`) or to use the one already in the `parser` (`FALSE`). This is rarely used in regular calls but for use internally when creating the TU parser and DefinitionContainer.

Value

An object of class DefinitionContainer.

Author(s)

Duncan Temple Lang

See Also

`resolveType`

`DerivedClassCode` *Generated code for a derived C++ class implementation with R functions*

Description

This represents the different pieces of generated code that defines and implements an extensions or derived class of a C++ class which allows the virtual methods to be implemented via R functions.

Slots

`className`: the name of the R and C++ class

`classDefinition`: the C++ definition of the new class, i.e. the C++ code that lists the fields, methods, inheritance, etc.

`rsetClass`: the R code to define the R class

`rfieldAccessors`: the R code to access an individual field of an instance of this newly derived class. This provides the method for `\[\[` so that `obj\[\["x"\]\]` retrieves the value of the C++ field named "x". This function knows about the available methods for this class and if there is no match, delegates to an inherited method so that the calls consult the ancestor classes.

`rconstructors`: R function(s) to create instances of these objects.

`methodNames`: a character vector giving the signatures of the methods and the names corresponding to the internal fields used to store the R functions that act as a method when using individual method fields.

`functionNames`:

`destructor`:

`RmethodIdRoutine`:

`methodImplementation`:

`methodNamesArray`:

`methodAccessors`:

`nativeClassConstructors`:

`ifdef`:

`callInherited`:

`sharedMethods`:

`sharedMethodsDef`:

`namesMethod`:

`protectedMethods`:

Author(s)

Duncan Temple Lang <duncan@wald.ucdavis.edu>

References

http://www.omegahat.org/RGCCTranslationUnit

freeVariables *Determine the free (non-local) variables in a C/C++ routine/method*

Description

This function examines the body of a C/C++ routine or method and finds the references to variables that are neither parameters or declared within the body.

This requires that the TU contain information about the body of routine/methods. This typically requires compiling the code with `g++`, even if the code is regular C.

Usage

```
freeVariables(def, nodes, vars = list())
```

Arguments

`def`	the node defining the routine (not the body)
`nodes`	the TU parser object providing the list of all the TU nodes
`vars`	

Author(s)

Duncan Temple Lang

gatherRegistrationInfo *Collect information about native routines for registering with R's dynamic lookup mechanism*

Description

There are two approaches to collecting the registration information for native routines in an DLL loaded into R. One is to take the existing C code and find all the possible routines that can be called by R and register those. The other starts with the R code and finds all the code that makes calls to C/C++ routines via the .C and .Call functions and then registers information only for the corresponding routines. Which you use depends on the style in which you write the R and C code, specifically which comes first.

The `gatherRegistrationInfo` function starts with the C/C++ code and reads one or more C/C++ source code file descriptions and collects the routines defined within those files that can be called from R via one of the native interface mechanisms into the different groups. This is information can then be used to generate R registration information for the routines that can be called via the .C, .Call and .External interfaces. (.External is not differentiated from the .Call at present.)

The `getRegistrationInfo` provides he other approach, starting with the R/S code. It reads code in the specified package and finds all the .C, .Call, .Fortran calls and identifies their routines. Then it reads all the TU files in the specified directory, typically the source directory for the package, and tries to match the R routine names to the routines in the C code. If there are mismatches, it uses fuzzy matching to try to suggest possible matches due to typos. Otherwise, if there are no matches, it

continues on generates code that can be inserted into a C source file and compiled with the package to perform the registration of the routines with R when the DLL is loaded.

The results of these functions can be passed to writeCode to output the resulting C code to a file for use in an programmatically generated interface.

Usage

```
gatherRegistrationInfo(fileName, tu = parseTU(fileName),
                       r = getRoutines(tu,  gsub("(.t00.tu|.tu)$", "", basename(
                       dir = dirname(fileName))

getRegistrationInfo(package, tu.dir,
                    foreignCalls = getNativeRoutineCalls(package),
                    routines = readRoutines(tu.dir),
                    robject = "SEXP")
```

Arguments

fileName	a character vector giving the name of one or more files. The file names are used to identify the .tu file and can be given as the name of the source file with or without the .c or .c++/.cpp/.cxx/.C extension. This is then used to find the .tu file and find the definitions of the routines within that particular file, i.e. as the files parameter for the getRoutines function. In other words, we use the name of the file to filter the nodes in the TU to those relating to this particular file. So rather than specifying, e.g., distance.c, or distance.c.tu, we use simply "distance" and the function determines the .tu file from this.
tu	rather than specifying the file name, one can pass the previously read array of translation unit nodes. This is useful if the TU file is very large, and/or you are doing additional operations on the contents of the file.
r	a list of the routines of interest. This is computed by calling getRoutines on the tu object. If however, the routines have already been identified, passing them directly avoids the overhead of reprocessing them.
dir	the directory in which the tu files are located. This parameer is convenient when one is specifying a collection of files within a different directory. Rather than having to paste the directory name to the file names, we can specify thes directory and file names separately.
package	this can be one of several types supported by the internal function getNativeRoutineCalls. These include a package name as a string, e.g. "XML", "stats4", or the package name with the "package:" prefix, e.g. "package:XML". One can also identify an already loaded package in the search path by giving the index of the package in the vector returned by search. Alternatively, one can pass it the name of one or more directories containing R source files and it will process each of the files ending in the extension .R, .S, .r or .s. Also, one can give it a list of function objects that have already been defined in R.
tu.dir	a string giving the name of a directory in which to find .tu files representing the translation units for the different source files in which we search for top-level routines.
foreignCalls	typically omitted, this is the information containing the collection of calls to the .C, .Call, etc. functions within the package or R source code specified by package.

`routines` the collection of C/C++ routine descriptions read from the translation unit(s). By default, we find all the .tu files in the directory specified by `tu.dir` and read those and extract the top-level routines that might be callable by R. If the TU file(s) has been read into R already, one typically wants to extract the routines from that and resolve them and pass those rather than repeating that potentially lengthy process.

`robject` the "word" in C for the native/internal data type representing an R or S object. This is here to allow us to use a more general name such as USER_OBJECT_ deployed in some macros used by Omegahat, and also to facilitate use with S-Plus.

Value

A list containing two elements `.C` and `.Call`. These are lists of descriptions of routines for the two different interfaces.

Author(s)

Duncan Temple Lang <duncan@wald.ucdavis.edu>

References

http://www.omegahat.org/RGCCTranslationUnit

See Also

`writeCode` `parseTU`

Examples

```
  files = paste(system.file("examples", package = "RGCCTranslationUnit"), c("arima", "dis
  f = gatherRegistrationInfo(files)

  o = generateRegistrationCode(f, "myPackage")

  writeCode(o, "native")
  writeCode(o, "namespace")

   # Note that we are starting with the R code.
 rfile = system.file("examples", "foo.R", package = "RGCCTranslationUnit")
 l = getRegistrationInfo(rfile, tu.dir = system.file("examples", package = "RGCCTranslati

   # write the code to the console
   # specify the name of the DLL as "duncan" to compute the name of the
   # init routine.
   # Also, turn off dynamic lookup and use registration only
writeCode(l, "native", dll = "duncan", dynamic = FALSE)

writeCode(l, "namespace", dll = "duncan")
```

GCC::TranslationUnit::Parser-class
Class "GCC::TranslationUnit::Parser" A convenience class for supporting an indexing method for the Perl array of nodes.

Description

This class is merely a convience class that is used to represent in R the result of parsing a translation unit from GCC using Perl tools. The class identifies a reference to a Perl array of nodes. We typically get this array via a call to `readTU` and then we access elements of the array by "following"nodes based on the INDEX field of another node. These INDEX values are strings that identify the nominal index of the element of interest. However, these need to be converted to the actual index by adding 1 to the integer value. The purpose of this class is to provide a method that handles indexing by these INDEX strings which does the necessary conversion. This makes the manipulation of the node paths easier at an interactive level.

Objects from the Class

A virtual Class: No objects may be created from it.

Extends

Class `"oldClass"`, directly. This is an old-style, S3 class that is used simply to provide a method for `[[`.

Methods

Currently, there are methods for accessing individual elements using the position or the node name. These take into account the extra first element in the Perl array that requires us to add one to the node name to get the actual index in the array.

The other two methods defined for this class are `lapply` and `sapply`. These hide the details how we loop over the Perl array. They might be better defined for PerlArray references directly in the RSPerl package.

Author(s)

Duncan Temple Lang <duncan@wald.ucdavis.edu>

References

http://www.omegahat.org/RGCCTranslationUnit

See Also

`readTU` The internal function `getIndex` which returns the actual integer index of a node along with its string identifier which is one less.

GCCNodeClasses *Perl parser translation unit node classes*

Description

S3-style classes to represent the nodes in the Perl translation unit (TU) parser

Objects from the Class

A virtual Class: No objects may be created from it.

Extends

Class "PerlHashReference", directly. Class "PerlReference", by class "PerlHashReference", distance 2. Class "oldClass", by class "PerlHashReference", distance 3.

Methods

freeVariables `signature(def = "GCC::Node::function_decl")`: ...

getCallGraph `signature(nodes = "ANY", defs = "GCC::Node::function_decl")`: ...

getInOutArgs `signature(routine = "GCC::Node::function_decl", nodes = "ANY")`: ...

Author(s)

Duncan Temple Lang

References

http://www.gnu.org/gccGCC

See Also

parseTU

Examples

```
showClass("GCC::Node::function_decl")
```

`generateInterface` *High-level function for creating R interface to C/C++ code*

Description

This is the high-level function that attempts to provide a simple interface for R users to generate code to allow her or other R users to invoke C/C++ routines and work with C/C++ data strucutres. It attempts to find all or a subset of the routines in a translation unit and generate an R interface to each along with the supporting functionality for converting and working with the data structures that are inputs or outputs to these routines.

The function expects a translation unit file (or parsed translation unit) and by default creates an R interface for each C/C++ routine in that TU. One can specify the routines of interest by name to identify a subset. The function determines which data structures are needed in this interface and generates R and C/C++ code for each these also. One can specify additional data types (e.g. enumerations that are not explicitly used in the routines) for the code generation. And one can also identify the global variables of interest by name.

Usage

```
generateInterface(tu, routines = character(), dataTypes = character(),
                  variables = getGlobalVariables(tu, files),
                  outfile = gsub(".[0-9]t.tu", "", basename(tu)),
                  includes = character(), language = NA,
                  omitPattern = character(), files = character(), typeMap = list
                  dllName = gsub(".R$", "", basename(outfile[1])), ...)
```

Arguments

`tu`	the name of the TU file, or the parsed TU object itself giving the collection of nodes in the TU.
`routines`	usually the names of the routines of interest which are found in the `tu` object. Alternatively, one can find and filter the routines prior to calling `generateInterface` (with `getRoutines`) and then resolve them via `resolveType`.
`dataTypes`	like `routines`, typically the names of the data structures of interest. One has to deal with duplicate matches and fetching the "wrong" data type when there are type definitions in effect. Alternatively, one can use `getDataStructures` prior to calling `generateInterface` and so filter and identify the data structures of interest. Code to interface to these data structures is generated as part of this function. You do not need to identify data structures that are used in routines, either as parameters or return types, as these are automatically processed and an interface for each type generated.
`variables`	
`outfile`	a character vector giving either the names of the R and C/C++ files into which the generated code will be written, or alternatively a single string which provides the base name of the file to which the appropriate prefixes will be added to generate the R and C/C++ file names.
`includes`	a character vector containing the files to `#include` in the generated C/C++ code so that all the native definitions are available. This is in the same form as for `writeCode` and `writeIncludes` and should have the names of the files enclosed in quotes or <>.

language	a string giving the native language of the C/C++ code
omitPattern	a regular expression which is used to identify routines which are to be discarded by matching the names of the routines with this regular expression. All matching names are omitted. This provides a convenient mechanism for saying "all but these routines" rather than having to enumerate a lengthy list of those to include.
files	a character vector which if specified is passed to getRoutines and getDataStructures to filter routine and data structure definitions. Only those routines and data structures defined within these files will be included.
typeMap	a list of mappings for customizing the generated code. Such elements provide information, for example, about how to convert a particular C-level data structure to R or vice verse.
dllName	the name of the file that will be compiled, linked and dynamically loaded into R containing the C-level interface to the native code. This is used in the generated code in the .Call expressions to identify the DLL specifically via the PACKAGE argument.
...	additional arguments passed on to createMethodBinding.

Value

A list with the following elements

source	a character vector giving the names of the files to which the code was written
routines	the names of the routines for which code was generated
dataTypes	the names of all the data structures for which interfaces were generated. This is the combination of both structures specified by the caller (via dataStructures) and those determined by the function to be needed in any of the interfaces to the routines.

Note

This current does not process C++ classes. This will be added later and can be done using other functions, e.g. getClasses and createMethodBinding and writeCode.

Author(s)

Duncan Temple Lang

References

The GNU compiler suite.

See Also

parseTU getRoutines getDataStructures getClasses

Examples

```
generateInterface("rlimit.cc.001t.tu",
                  c("getrlimit", "setrlimit"),
                  dataTypes = "__rlimit_resource",
                  outfile = "rlimit",
                  includes = c("<sys/time.h>", "<sys/resource.h>", '"RConverters.h"'),
```

```
                    language = "C")

v = generateInterface("rlimit.cc.001t.tu", , ,
                      outfile = "bar",
                      includes = c("<sys/time.h>", "<sys/resource.h>", '"RConverters.h"'
                      language = "C", omitPattern = "64", files = "resource.h")

v = generateInterface("tmp.c.001t.tu", , ,
                      outfile = "bar",
                      includes = c("<sys/time.h>", "<sys/resource.h>", '"RConverters.h"'
                      language = "C", omitPattern = "64")
```

`generateStructInterface`

Generate R and C interface code to create, access and set aspects of a C structure

---

Description

This function is a high-level function that operates on the description of a single C level structure definition and generates the R and C code to interface to instances of that structure. It generates a parallel R class that has the same slot names as fields in the C structure with the equivalent types in R. Additionally, it creates a class that can be used to hold a pointer to a C-level instance of the C structure via an `externalptr` in R. Also, it generates R functions and corresponding C routines to create new instances of this C-level data type and to access the fields individually and also to copy an entire C-level instance to its R equivalent. In other words, it allows the R user to work with the structure either in R or in C and to access all parts of it.

Usage

```
generateStructInterface(type, classDefs, typeMap = list(),
                            defaultBaseClass = "RC++StructReference")
createCopyStruct(def, className = def@name, isClass = FALSE, typeMap = list())
```

Arguments

`type`	an object representing the `resolved` type definition of a C-level structure. In other words, it should be an object created by a call to `resolveType` (either implicitly or explicitly) and either a `StructDefinition` or a `TypedefDefinition` corresponding to a `typedef` in R which leads to a `StructDefinition`.
`typeMap`	an optional type mapping specifying the correspondence between the R and C-level data types and how they can be converted from one to another. This is used to define the types of the R slots corresponding to the fields in the C structure and how to convert between them. This is optional and the default mappings will work in pretty much all cases. This allows one to customize the mappings with contextual information/knowledge.
`classDefs`	
`def`	the description of the class/struct definition.
`className`	the name of the struct/C++ class
`isClass`	a logical value indicating wehther this is a C++ class or a regular C `struct`.

`defaultBaseClass`
: the name of the R class which is used as the base class for the class that is a reference to this struct/C++ class. This allows the caller to use a different base class with a different representation or methods.

Value

A list of class `CStructInterface` which can be passed to `writeCode`. The list has the following components:

`generic`

`rFunctions`

`cRoutines`

`clasDefs`

`coerce`

`newInst`

`freeInst`

Author(s)

Duncan Temple Lang

References

The GCC Translation Unit

See Also

`createInterface`

`getAllDeclarations` *Find top-level translation unit nodes representing different types of entities/constructs within the source code.*

Description

These functions are useful, top-level functions for finding the different entities within the source code via the translation unit array. These find things such as global variable declarations, function and data structure declarations and classes. These identify the nodes of interest and then we can process these nodes to turn them into more directly useful and high-level information such as class and function descriptions.

These functions use `nodeIterator` to find the nodes of interest, using different filter functions.

`getFunctions` and `getRoutines` are identical. They are provided as I use the term routines to refer to what many people call functions which are in native code. This draws a distinction between R and C/C++ "functions" and gives us a clearer vocabulary that is also more consistent with computer science definitions.

`getEnumerations` is slightly different from the other functions documented on this page. While its return value is, like the others, a list of indices identifying the target nodes in the translation unit, the way in which it traverses the nodes is different. This function loops over all of

the nodes with the translation unit lookng for enumeration declarations, i.e. Perl nodes of class `GCC::Node::enumeral_type`. It does this because enumerations may be defined outside of the regular declarations. Since we loop over the entire array of tu nodes, this is done directly in Perl to avoid the overhead of transferring control between R and Perl for each element.

Usage

```
getAllDeclarations(dcls, files = character(0), dropArtificial = TRUE)
getGlobalVariables(dcls, files = character(0), static = TRUE,
                     resolve = FALSE, checkSourceFile = checkSource, ...)
getFunctions(dcls, files = character(0), static = FALSE,
               checkSourceFile = checkSource, ...)
getRoutines(dcls, files = character(0), static = FALSE,
              checkSourceFile = checkSource, ...)
getClassNodes(dcls, files = character(0), ignoreClasses = character(0),
               ..., validateSource = checkSource)
getEnumerations(dcls, files = character(),
                  extensions = c("c", "cpp", "C", "h"))
```

Arguments

`dcls`	the starting node from which to iterate over the chain.
`files`	a character vector. If this is specified, the filters only return information from nodes that have a source attribute (srcp) which corresponds to an element in the `files` argument. The source attribute is modified to remove the line number and extension. Note that for `getEnumerations`, this must be "" and not `character(0)` as it is passed to Perl.
`ignoreClasses`	this itself is currently ignored!
`...`	any additional arguments that are passed on to the `nodeIterator` call which in turn are passed on also to the calls to the `op` function within `nodeIterator`.
`dropArtificial`	a logical value indicating whether to omit any nodes that are "artificially" created by the compiler. In general, these are good things to omit.
`static`	a logical value indicating whether to omit routines that are declared as static and so not visible outside of the specified files or if `FALSE` to include them in the result.
`resolve`	(logical) controls whether the resulting TU nodes are then resolved using `resolveType`.
`checkSourceFile,checkSource`	
`validateSource`	
`extensions`	

Value

A list with elements corresponding to the processed nodes of interest. The nature of those elements depends on the type of node. They all however identify the index of the node.

Author(s)

Duncan Temple Lang <duncan@wald.ucdavis.edu>

References

http://www.omegahat.org/RGCCTranslationUnit

See Also

readTU getBaseClasses

Examples

```
my = parseTU(system.file("examples", "ABC.cpp.tu", package = "RGCCTranslationUnit"))

 # All the declarations found when processing this file, including the
 #  '#included' files via the pre-processor.
k = getAllDeclarations(my)

 # Just the declarations defined in ABC.cpp, ABC.h, ...,
 # i.e. with a source field starting with "ABC."
k = getAllDeclarations(my, "ABC")

my = parseTU(system.file("examples", "myFun.cpp.tu", package = "RGCCTranslationUnit"))
d = getAllDeclarations(my, "myFun")
names(d)

   # Get the node types
types = sapply(d, function(i) class(my[[i]])[1])
table(types)


 # Focusing on the global variables only, we can get them from types
 # or by querying the TU for them alone.
vars = getGlobalVariables(my[[4]], "myFun")
vars
#XXX  fails on a pointer to a function_type (no method for this type) in a recursive call
# lapply(vars, function(x) resolveType(my[[ x ]], my))
```

---

getBaseClasses *Compute names of base or ancestor classes*

Description

This function determines the names and nodes of the different classes from which the target class inherits. These are the base or parent classes.

By default, the function returns the parent classes, i.e. those classes which this class directly extends. If there is more than one parent class, this indicates multiple inheritance.

Alternatively, via the `recursive` argument, we can compute the entire ancestry of base classes. In other words, we can get the parents, their parents, and so on.

For any base class, the function can return either its identifier in the translation unit (i.e. the node index), or the "raw" information about the class. This is a reference to a Perl hash table which

contains just three elements: access, virtual and class. The first two indicate whether the class is extended in a public, protected or private manner, and whether the parent class is virtual or not. The third field, class, gives information about the class definition via a `GCC::Node::record_type` object.

Usage

```
getBaseClasses(node, getReferences = FALSE, recursive = FALSE)
```

Arguments

`node`	the node in the translation unit array that defines the method. This is typically one of the elements references from the 'fncs' attribute of the class' `record_type` node.
`getReferences`	a logical value which controls whether the raw array of class information is returned (`TRUE`) or, by default, just the indices of the translation unit nodes which define the parent classes are returned.
`recursive`	a logical value, which if `TRUE` causes the function to recursively process the parent nodes and find their parent classes, and then their parent classes, and so on.

Value

If `getReferences` is `TRUE`, a Perl array is returned. This has as many elements as there are direct base or parent classes for this class definition. Each element is a `GCC::Node::record_type` in the parsed TU collection, and this gives the class definition.

Rather than simply being a Perl array, the return value when `getReferences` is `TRUE` is an object of class `BaseClassInfo`. This is a simple S3 class label that is used to provide convenient access to the elements of the Perl array by the class name each element defines.

Author(s)

Duncan Temple Lang <duncan@wald.ucdavis.edu>

References

http://www.omegahat.org/RGCCTranslationUnit

See Also

`readTU` `getAllDeclarations`

Examples

```
  # Find the classes in ABC.h/ABC.cpp
my = parseTU(system.file("examples", "ABC.cpp.tu", package = "RGCCTranslationUnit"))
k = getClassNodes(my)

names(k)

getBaseClasses(  k$B  )
getBaseClasses(  k$C  )
```

```
 # get the entire ancestry of C
getBaseClasses( k$C, recursive = TRUE)
 # Just the immediate classes.
getBaseClasses( k$C, recursive = FALSE)

  # return the class definiiton nodes for the base class
  # and not just the name.
r = getBaseClasses( k$B, getReferences = TRUE)

names(r)
.PerlLength(r)

  # Explore the class Other via its node.
  # We could do this from k[["Other"]] also
names(r[["Other"]])
r[["Other"]][["access"]]
r[["Other"]][["virtual"]]

def = r[["Other", convert = FALSE]][["class", convert = FALSE]]
names(def)
```

`getCallGraph` *Compute call graph for a C/C++ routine or method*

Description

This actually works on arbitrary nodes within the translation unit parser array of nodes. So it can be used for examining code blocks, e.g. body of an if statement or a while loop.

Usage

```
getCallGraph(nodes, defs = getRoutines(nodes), visited = new.env())
```

Arguments

`nodes`	the parser or array of nodes, used to lookup the references to other nodes within the `defs` object.
`defs`	for the caller, this is the routine returned from `getRoutines`. In fact, it can also be any node in the parser array, i.e. a GCC::Node.... type.
`visited`	this is not intended to be supplied by the top-level caller, but rather is passed between the recursive calls to store information about which nodes have already been visited and so are not to be processed again should there be another node that refers to it.

Value

A named vector giving the names of the routines that were found to be called within the routine/language node along with the position in the parser array of nodes so that one can quickly identify the actual declaration/definition.

Note

This does not handle invocations via "function pointers" in the C code. It can be made to understand them, but not know which routine will be called. That is a run-time decision.

Author(s)

Duncan Temple Lang

References

c-tree.texi in the gcc source distribution.

See Also

parseTU

Examples

```
filename = system.file("examples", "phast", "msa.c.tu", package = "RGCCTranslationUnit")

p = parseTU(filename, language = "C")
routines = getRoutines(p, "msa.c")

calls = getCallGraph(p, routines$msa_new_from_file)
names(calls)
counts = table(names(calls))
```

---

getClassMethods *Get descriptions for all methods in a C++ class*

Description

This method iterates over the nodes within the translation unit array that are referenced from the class definition node via the 'fncs' attribute in the node. It creates a basic description of the method, capturing the node identifiers for the parameters, return type, etc. To make use of these, they must be resolved at a later point. (See resolveType.)

Usage

```
getClassMethods(def, accessMode = c("public", "protected"), existingClasses = li
                dropArtificial = TRUE, dropOperators = TRUE, ...)
```

Arguments

def	the node in the translation unit that gives the definition for the class of interest.
accessMode	a vector identifying whether public, protected and/or private methods should be processed. If this is non-empty, the access attribute of a method node is compared to the elements in this list and if it matches it is included in the return. If this is empty, all method nodes are processed.
existingClasses	currently ignored. This is intended to be a repository or catalog of resolved type definitions to avoid them being re-processed.
dropArtificial	a logical value indicating whether to discard methods that are identified as "artificial", i.e. generated by the compiler rather than explicitly defined in the code.

dropOperators
: (logical) include or omit any C++ operator methods such as `operator() +`.

`...`
: additional arguments passed to `nodeIterator` and hence possibly onto the function that processes each visited node.

Value

A list with an element for each method declared within the class. The names of the methods are used as names for the list elements.

Each element describes the corresponding method and is of class `NativeClassMethod`, a simple S3-style class. Each such element has the following fields:

`returnType`
: the index of the node in the translation unit the defines the type of the return value from the method.

`parameters`
: a list of parameters (which may have names for the elements), each giving the index of the nodes defining the parameter.

`index`
: the index of the node defining this method

`name`
: a character vector giving the name of the method.

Author(s)

Duncan Temple Lang <duncan@wald.ucdavis.edu>

References

http://www.omegahat.org/RGCCTranslationUnit

See Also

`readTU` `getClassNodes`

`resolveType`

---

getCppDefines *Compute the C/C++ pre-processor macro definitions*

Description

The `getCppDefines` function attempts to get the definitions of the pre-processor definitions from one or more header/source files by using GCC's ability to emit the table of macro definitions. It compares the collection of defined symbols to that without the source file (but with specified system header files) so as to find the ones that are actually defined within the user-level source files and not the system definitions. These can then be processed further to organize them into different categories using, e.g. `processDefines` and `filterMacros`.

Usage

```
getCppDefines(fileName = character(), cppFlags = "", cpp = "g++",
              flags = "-E -P -dM", sysIncludes = c("stdlib.h", "stdio.h"))
```

Arguments

`fileName`	the name of the C/C++ source file which is to be passed to the pre-processor. This is the file containing the code for which we want to extract the pre-processor definitions.
`cppFlags`	any extra C/C++ pre-processor flags needed for processing `filename`. These can be literal strings giving the flags or alternatively, a shell command that computes the flags within the execution of the shell command to run the pre-processor. Such dynamic flags should be enclosed within double quotes with the command in back-quotes, e.g. `"`wx-config --cflags`"`. And this is platform-specific, working on Mac OS X and UNIX/Linux.
`cpp`	the name of the C/C++ pre-processor. By default, we use the GCC C++ compiler and instruct it to only run the pre-processor phase of the compilation via the `flags` arguments. One can specify an executable and flags here as the strings are pasted together to form the command. By separating the flags and the executable, we make it easier to specify a different executable, e.g. in a different location/directory than the default.
`flags`	additional flags or command line-arguments to be passed to the pre-processor command (i.e. `cpp`). These control how the processor is invoked relative to executable which may be the regular compiler.
`sysIncludes`	a character vector which gives the names of the system-level include files which are to be treated as the baseline. These are used by first computing all the macro definition symbols that occur in these files (and the compiler/pre-processor itself) and removing these from the symbols that are found when we apply the pre-processor to the the user-specified source file, i.e., `fileName`.

Details

This uses the GCC suite's ability to dump the table of pre-processor symbols via command line arguments.

Value

A character vector giving the set of macro definitions as lines of the form `"#define SYM definition"`. Macros that have multi-line definitions will have multiple entries in this vector with the subsequent lines of the definition being successive elements in the result. These are manipulated and pulled into single definitions using `processDefines`.

Author(s)

Duncan Temple Lang

References

The GCC pre-processor

See Also

`processDefines`

getDefineConstants *Find the constants defined via the pre-processor*

Description

This attempts to find the definitions of literal constants that are created by the C/C++ pre-processor and not in the C/C++ language. This finds entries such as `#define FOO 1` or `#define STR "my string"`. It does not handle conditional definitions, i.e. within `#ifdef` statements.

This works by making a system call to grep on the files in the target directory. (It does not work recursively, but could easily do so if that was useful rather than excessive.) It discards lines that start with a C++ comment or with #include. Then it delegates the identification of literal constant definitions to the internal function `processDefines` which looks for all lines that start with #define.

A "better" approach is to use `getCppDefines` which works directly from the output of the pre-processor.

Usage

```
getDefines(dir = character(), pattern = ".*", files = "*",
           class = c("TopLevelConstants", "DefineConstants"),
           removeDuplicates = TRUE, getSkipped = FALSE)
```

Arguments

`dir`

`pattern`

`files`

`class`

`removeDuplicates`

`getSkipped`

See Also

`getCppDefines`

getExportNames *Get names of R objects to be exported from generated bindings*

Description

This generic function is used to collect the names of the R functions, classes and generic functions that are to be added to the NAMESPACE file of a generated package that includes these machine-generated bindings.

Usage

```
getExportNames(obj)
```

Arguments

obj	the object from which to get the generated binding information. Currently, this is the object returned from createClassBindings, but will be extended to handle higher-level aggregates for an entire file and its declarations and collections of files making up a native code library.

Author(s)

Duncan Temple Lang <duncan@wald.ucdavis.edu>

References

http://www.omegahat.org/RGCCTranslationUnit

See Also

createClassBindings

---

getFields *Find the TU nodes for fields or slots in a class or struct or union*

Description

This function follows the chain of translation unit nodes to collect the nodes defining fields within a C/C++ container object such as a struct or union or a C++ class. It is called with the TU node identifying the class or struct/union definition, and the function then traverses the linked nodes using nodeIterator. It can distinguish between public and/or protected fields which can be useful in different contexts, e.g. when creating regular accessors or generating derived classes whose methods can "see" protected fields.

Usage

```
getFields(def, access = c("public", "protected"), existingClasses = list(),
          artificial = FALSE)
```

Arguments

def	a TU node identifying the container definition, i.e. a GCC::Node::type_decl or GCC::Node::record_type or a node identifying a field or variable declaration within a container. Passing a field or var declaration causes the function to iterate over the fields from that point on. This could be useful if one want to skip over certain fields, e.g. the first field in a C struct when it is being used as a primitive inheritance technique. This value is passed to field.decl to find the first field.
access	a character vector specifying whether we are interested in public and/or protected fields. These are matched against the access attribute within the processed TU field/var nodes. This uses partial matching and so, for example, "pro" or "pri" would suffice. If this is either the empty character vector (i.e. character()) or NA, all fields are returned. This is equivalent to c("public", "protected", "private").

existingClasses
: unused

artificial
: a logical value indicating whether to include (TRUE) or discard artificial fields that are generated by the compiler and not explicitly part of the structure as programmed.

Value

A list containing the translation unit nodes corresponding to the fields of interest.

Author(s)

Duncan Temple Lang

See Also

parseTU.Perl nodeIterator as.field.decl

Examples

```
library(RGCCTranslationUnit)
p = parseTU(system.file("examples", "shapes.cc.t00.tu", package = "RGCCTranslationUnit"))
klasses = getClassNodes(p, "shapes")

    # Get all fields in the Shape class
 getFields(klasses[["Shape"]], character())

    # Get the public and protected fields in the Shape class
 getFields(klasses[["Shape"]])

 getFields(klasses[["Circle"]])

    # all fields, i.e. the private radius
 getFields(klasses[["Circle"]], character())
```

---

getInOutArgs *Determine which parameters are also output values*

Description

This function (and the collection of supporting methods for recursive processing of the nodes) attempts to determine whether any of the mutable parameters, i.e. pointers or references, are actually modified with the body of the routine. If they are, these are probably out variables that we would want to return to the R user in a call to the specified routine.

This is incomplete at present and more a working prototype for people to add methods for new cases.

It does not try to determine inout variables, but just out. An inout parameter is one whose value is also used as an input to the code, i.e. is on the right hand side or in a call to another routine within the code and not just assigned a value.

Usage

```
getInOutArgs(routine, nodes, params = list())
```

Arguments

routine	the resolved routine of interest. In the future, we will add support for working with the simple routine description or node returned from getRoutines, i.e. one will not have to resolve the routine, although this is not typically a hardship.
nodes	the parser, i.e. array of nodes. This is the return value from parseTU and is a reference to a Perl object of class GCC::TranslationUnit::Parser.
params	this is not usually specified by the top-level caller but calculated from the definition of the routine and passed to the recursive calls. This is the character vector giving the index or identifier of the parm_decl nodes corresponding to the mutable parameters of the routine.

Details

We need to enhance this to deal with assignments to local variables which are themselves modified. Also, we need to add a mechanism to identify the routines that are called with any of these parameters and the identity of the parameters so we can recursively determine things.

Author(s)

Duncan Temple Lang

See Also

createMethodBinding getCallGraph

Examples

```
filename = system.file("examples", "inout", "inout.c.tu", package = "RGCCTranslationUnit
o = parseTU(filename)
rr = getRoutines(o, "inout.c")
routines = resolveType(rr, o)
getInOutArgs(routines$foo, o)
getInOutArgs(routines$bar, o)
```

---

getNativeDeclaration

Compute C/C++ declaration for specified variable

Description

This function is responsible for creating the text that declares a C/C++ variable of the specified type so that it can be used as a parameter in a C/C++ routine or as a local variable within the routine. It combines a type and a symbol name to create the declaration code. It can optionally put a ; at the end of the code.

Usage

```
getNativeDeclaration(name, v, variableNames, addSemiColon = TRUE, const = NA)
```

Arguments

name	the name for the variable or parameter
v	a description of the type of the variable being declared. This is typically obtained by resolving a type definition from the translation unit using resolveType but can be created manually.
variableNames	the names of other known variables in the routine definition which could be used to avoid conflicts.
addSemiColon	a logical value indicating whether to add a ';' at the end of the generated declaration code. When generating parameter declarations for a routine, we don't want the ;. But for local variables, we do.
const	this is used in some of the methods to qualify the declaration as being a constant/immutable "variable". This information can also be determined in many cases from the type v. If this is NA, the value is determined from v. Otherwise, it should be a logical value with TRUE indicating to add a const qualifier to the generated code declaration.

Value

A string (character vector of length 1) giving the declaration for the code.

Author(s)

Duncan Temple Lang

---

getNodeSource *Source file and line number information for a TU node*

Description

This returns the information in the TU node about the source file and line number in which the node is associated.

Usage

```
getNodeSource(node)
```

Arguments

node	a node from the parsed translation unit (tu

Value

A string of the form filename:num

Author(s)

Duncan Temple Lang

See Also

parseTU

getRClassDefinitions
Compute R classes corresponding to a C++ class

Description

This function takes a translation unit node defining a class and then computes information about it to define an R class corresponding to that C++ class. It provides the R code to define this class as well as providing information about the ancestor classes. When working in recursive mode, the function can generate the R code to define classes for the entire hierarchy of ancestor classes.

Usage

```
getRClassDefinitions(node, recursive = FALSE, className = getNodeName(node),
                     defaultBaseClass = "RC++Reference")
```

Arguments

node	a translation parser node which defines a C++ class. This has class GCC::Node::type_decl and is a reference to a Perl object.
recursive	a logical value indicating whether to process the node and its ancestor classes recursively or just this leaf node by itself. If this is TRUE, the function returns a list with information about each of the classes in the ancestor chain.
className	the name to use for the R base of the class name. Note that "Ptr" is appended to this.
defaultBaseClass	the name of the base R class to use in the R class definition if the specified C++ class does not have any ancestors.

Value

If recursive is FALSE (the default) this returns an object of S3 class ClassDefinition which has fields named:

definition

baseClasses	a named character vector giving the names of the classes from which this C++ class is derived. The names for this vector are the node identifiers in the translation unit of the corresponding class nodes for these parent classes.
ancestors	a named character vector giving the name of the C++ class and the name of that element is the corresponding R class name
className	the name of the class being defined

If recursive is TRUE, the function returns a list with elements of class ClassDefinition as described in the previous paragraph.

Author(s)

Duncan Temple Lang

References

The GCC compiler suite

See Also

getClassNodes setClass

`getRTypeName` *Name of R type corresponding to native type*

Description

This function is used to find the name of the R type corresponding to the native type description.

Usage

```
getRTypeName(type, typeMap = list(), ...)
```

Arguments

`type`	the native type object
`typeMap`	a caller-specifiable list of native type and conversion information. This can be used to override the default methods and allows one to customize the the code generation process for particular types that one wants to deal with in special ways.
`...`	additional parameters for methods

Value

A string.

Author(s)

Duncan Temple Lang <duncan@wald.ucdavis.edu>

`getVariables` *Find the translation unit nodes corresponding to variable declarations*

Description

This function searches a translation object in R for nodes that correspond to variable declarations. This returns the raw nodes. These can then be resolved to get complete descriptions of the variables and their data types.

Usage

```
getVariables(dcls, addNames = TRUE, ...)
```

Arguments

dcls	the translation unit object as returned by parseTU.
addNames	a logical value indicating whether to compute the names of the variables and put them on the resulting list
...	additional parameters for methods

Value

A list whose elements are objects of class GCC::Node::var_decl.

Author(s)

Duncan Temple Lang

See Also

parseTU resolveType

Examples

```
tu = parseTU(system.file("examples", "struct.c.t00.tu", package = "RGCCTranslationUnit")
v = getVariables(tu)
names(v)

vars = lapply(v, resolveType, tu)
```

---

nodeIterator *Walk and process the list of linked/chained nodes.*

Description

This function is the workhorse for processing the nodes in different ways. This function takes care of looping over the sequence of nodes and uses a predicate function type to determine if the function is of interest.

Usage

```
nodeIterator(node, op, type = TRUEp, verbose = FALSE, addPrefix = FALSE)
```

Arguments

node	the node from which to start the iteration. This can be an arbitrary node in the array of translation unit nodes.
op	the function to process the node. If this returns NULL, the value is discared. Otherwise, it is added to the list of results. An attempt is made to compute the name of the give node to identify the element in the resulting list. This function can of course work on other nodes aside from the one it is passed. And it can store information in its own environment rather than relying on the construction of the appropriate list by nodeIterator. However, this is not necessary.

type	a function or the class name(s) of the type of node to process. If this is a character vector of class names, nodeIterator creates a simple function that calls inherits. This is simple "static" type checking of the nodes. A user can provide a function that checks the contents of the node, dynamically. The function is called for each node in the sequence. If it returns FALSE, the iterations stop and the result returned. So this checks whether the processing is complete for the given sequence and allows for termination of the looping over the chained nodes before the end of the chain is reached.
verbose	a logical value with TRUE indicating that information about what nodes are currently being processed, e.g. the source and desition or to and from nodes, is to be displayed on the console as the iterations proceed.
addPrefix	(logical) passed to getNodeName when computing the name of the node to use in the names of the returned list. This controls whether the C++ namespace of the enclosing scope is included (TRUE) in the name or dropped (FALSE).

Value

A list with an element for each node processed. For each such node, the name of the element in the list is taken from the "name" attribute of the node, if available.

Author(s)

Duncan Temple Lang <duncan@wald.ucdavis.edu>

References

http://www.omegahat.org/RGCCTranslationUnit

See Also

getAllDeclarations getGlobalVariables getFunctions getClassNodes

These functions are all implemented using nodeIterator.

Examples

```
tu = parseTU(system.file("examples", "myFun.cpp.tu", package = "RGCCTranslationUnit"))

nodeIterator(tu[[4]], function(node) node[["INDEX"]])
```

---

parseTU *Read a Translation Unit file into memory.*

Description

This function provides a high-level interface to the Perl tools to process a translation unit graph and bring it into memory. The result is a Perl array of GCC::Node node types that describe the entities within the original source code. This array can be indexed by position from within R and the individual nodes can be processed in various ways.

This function tolerates mis-specified file names allowing one to specify the name of the C/C++ source file or one with a ".tu" extension instead of the ".t00.tu" extension generated by more

modern/up-to-date versions of GCC. In other words, if the TU file is named `abc.cc.tu`, the caller can specify the file name as `abc.cc`, `abc.cc.tu` or `abc.cc.t00.tu`. This forces the function to determine which extension is being used and not the caller.

One can set the source language as "C" or "C++" when creating the parser with `parseTU`, but `setLanguage` also allows one to set it after the parser has been created and the parsing performed.

`parseTUOriginalTree` reads the type of file generated with the gcc/g++ command line flag -fdump-tree-original-raw which results in a file with multiple, separate TUs for each routine in the original source. Each TU has its own set of nodes, numbered starting from 1 and so must be treated separately. This function returns a list of objects obtained by calling `parseTU` on each of the separate sub-TUs using `asText = TRUE`.

Usage

```
parseTU(filename, language = NA, typedefs = NULL, asText = FALSE)
setLanguage(parser, language = NA)
parseTUOriginalTree(fileName)
```

Arguments

`filename, fileName`	the name of the file containing the translation unit.
`language`	a string such as "C" or "C++" which tells the parser and, more specifically, code that uses the contents of parsed nodes that the original code from which the translation unit was generated was actually C code even if g++ was used to generate the tu file. This influences how, for example, structs are resolved, being `StructDefinition-class` for C code rather than `C++ClassDefinition-class` for C++ code.
`typedefs`	an optional `DefinitionContainer` object that can be passed by the caller to initialize the `DefinitionContainer` within the parser object and which is used to resolve nodes.
`parser`	the parser object returned from a call to `parseTU`.
`asText`	a logical value indicating whether the `fileName` is the name of a file containing the translation unit (`FALSE`) or if `fileName` is the actual contents of the TU (`FALSE`) as read from a file by an R command (e.g. `readLines`)

Details

This uses the Perl GCCTranslationUnit module to read the translation unit and uses the `RSPerl` package to initiate this action and provide access to the resulting array.

Value

A reference to a Perl array which is of class `GCC::TranslationUnit::Parser`. This S3-style class has methods in R that make it easier to work with and treat more like a regular R list. It allows indexing by position or node name (taking into account the first element) , and provides lapply/sapply methods for looping over the elements and applying a function to each node. See `GCC::TranslationUnit::Parser-class` for more information.

Author(s)

Duncan Temple Lang <duncan@wald.ucdavis.edu>

References

http://www.omegahat.org/RGccTranslationUnit

See Also

getAllDeclarations getGlobalVariables getFunctions getClassNodes nodeIterator

Examples

```
tu = parseTU(system.file("examples", "myFun.cpp.tu", package = "RGCCTranslationUnit"))
tu = parseTU(system.file("examples", "foo.c", package = "RGCCTranslationUnit"), languag

  # Number of nodes in the translation unit
.PerlLength(tu)
class(tu)
getRoutines(tu)
```

`processDefines` *Organize and filter the raw macro definitions into categories.*

Description

This function `processDefines` processes the lines of macro/pre-processor definitions (typically obtained by a call to `getCppDefines` and organizes the lines by combining multiple lines definitions into single macros and then does an initial filtering to remove the basic definitions and duplicates. Finally, the function filters the definitions according to regular expressions and other, context specific "rules" and arranges the macros into different categories identifying them as simple flags, constants, macros whose bodies need to be calculated in C code, parameterized macros (i.e. accepting arguments) and macros that are to be ignored, e.g. barriers for avoiding recursive inclusion of a header file (`#ifndef FOO_H. #define FOO_H ... #endif`.

`filterMacros` is the default filtering mechanism used to identify macros that we don't want to map to the R interface and modify any that need to be changed for mapping into R. Instead of a single filter function, one can pass a list of these filter functions and they will be called in turn with the output from the previous function so that we progressively filter the collection of macro definitions.

Usage

```
processDefines(lines, class = c("TopLevelConstants", "DefineConstants"),
               headerIfDefPattern = "(_H_{,2}|_H_BASE_)$",
               keepFlags = length(headerIfDefPattern) && nchar(headerIfDefPatter
               filter = filterMacros, ..., tu = NULL)

filterMacros(qq, namePatterns = c("^SQL", "pthread"),
             valuePatterns = c(),
             globalConstants =
                computeGlobalConstants(tu, getGlobalVariables(tu, files), defs =
             enumDefs = computeGlobalEnumConstants(tu,
             getEnumerations(tu, files), defs = typeDefs),
             tu,
```

```
                typeDefs = DefinitionContainer(tu),
                files = character())
```

Arguments

lines	a character vector giving the lines from the pre-processor defining the macros, including multi-line macros whose contents are spread across multiple (contiguous) elements of this vector. Typically this is the output from a call to `getCppDefines`, but other approaches to obtaining these lines are possible, e.g. grep.
class	a character vector, currently ignored, but intended to specify the S3 class label of the result of this call.
headerIfDefPattern	a regular expression which is used to identify macro names which are used simply to avoid repeated #include's of header files. This can be a patterned regular expression such as the default, or if one knows the header files and their barrier names, one can specify a literal OR'ed regular expression of the form, e.g., `"(A_H|B_H)"`. This is only used if `keepFlags` is `TRUE`.
keepFlags	a logical vector which, if `TRUE`, controls whether the function endeavors to process the macro symbols which are simply flags or whether it adds them to the list of ignored macros.
filter	a function or a list of functions. Each function is called with the processed results and can perform further reductions and re-organiztations. When a list of functions is provided by the caller, the output of one function is passed as the input to the next function for continued filtering and the result is the object returned by the last of these functions. This allows one to use a collection of separate filters to do all the filtering that is needed.
qq	
namePatterns valuePatterns	
globalConstants	
enumDefs	
tu	
typeDefs	
files	
...	

Value

The form of the result depends on the filter function(s) in the call. However, the default mechanism produces a list with five entries:

parameterizedMacros	macros which take arguments
macros	simple macros which have constant/literal values.
flagDefines	simple macros definitions that are used merely to indicate a state that is TRUE or ON. These all have value `"TRUE"`.

`calculate`	macro definitions whose values are calls in the R sense, e.g. `A | B` or even -1 since that is parsed in R as a call to -. These can all be processed and read into R so the fact that some literals are treated this way is not important.
`ignored`	macros which are not understood by R or deliberately excluded.
`...`	passed to the call to `filter`, if that is non-`NULL`.

Author(s)

Duncan Temple Lang

References

The GCC pre-processor

See Also

`getCppDefines` `filterMacros`

RC++Reference-class

Basic class for representing external C++ objects

Description

These are the basic, top-level R classes for representing references to C/C++ objects that are maintained as external pointers.

Objects from the Class

Typically, this class is not used directly. Rather, R classes are defined that extend this one and mirror the corresponding C++ class. The purpose of this class is to provide methods for accessing fields and methods in the underlying C++ object via the `$` operator. Instances of this class are typically created directly in the C/C++.

This class and related code can be used directly by programmers. However, they are offered here as support for machine generated code. They can be used within the R-SWIG interface, as well as other approaches to interfacing with C/C++ code.

Slots

`ref`: Object of class `"externalptr"`. This is the value of the memory address of the actual C/C++ object.

`classes`: Object of class `"character"` . This vector stores the names of all the ancestor classes for an instance of this specific class. This information is used in C/C++ code when dereferencing the address stored in the `ref` slot. It checks that this reference is compatible with the target class by comparing the target name to each of the class names in this slot.

Methods

$ `signature(x = "RC++Reference")`: this provides the basic syntactic-sugar for invoking methods and accessing fields in the underlying C++ object using the form `obj$method(arg1, arg2, .....)` and `obj$field`

Author(s)

Duncan Temple Lang <duncan@wald.ucdavis.edu>

References

http://www.omegahat.org/RGCCTranslationUnit

`RCode` *Functions for creating representations of R code as text*

Description

These functions allow us to represent R objects which represent various types of R code, functions and methods. These also simplify indenting the code.

Usage

```
RCode(..., .txt = unlist(list(...)), indent = "", class = "RCode")
```

Arguments

`...`	character strings giving the text for the code
`.txt`	an alternative way to provide the strings for the code
`indent`	the string to prefix the text, i.e. indent.
`class`	the type of object. This allows us to use the same function to construct different types of objects

Value

An instance of class `class` containing the code as a single string concatenated togther with a new line character (\n).

Author(s)

Duncan Temple Lang

Examples

```
RCode("if(is.null(x))", "   stop('x is null')", "x[[2]]")
```

`readDependencies` *Read source code file dependencies on other files*

Description

This function is used to post-process the output of a call to the GCC compiler (gcc or g++) to generate the list of files on which a particular source file depends, i.e. includes either directly or indirectly. This information is then used to filter the symbols, routines, data structures, etc. in which we are interested by limiting them to files typically other than these or by matching these to ones in a known directory of interest.

This function does not perform the call to the compiler to generate the dependencies. Instead

Usage

```
readDependencies(filename)
```

Arguments

`filename`	the name of the file to which the dependency information was written in the call to the GCC compiler, or the output itself, i.e. the raw lines emitted by the compiler.

Value

A character vector giving the fully-qualified names of the included files and the names of the elements in this vector are the corresponding base names of these files. Since the TU output discards the directory information, we return this base name form for easy comparison but maintain the directory information for completeness. So often we use the `names()` of the result for use with TUs. Additionally, the name of the file from which the dependencies were computed, i.e. the source file passed to the compiler, is available via the attribute `"source"`.

Author(s)

Duncan Temple Lang

References

The GCC compiler suite

See Also

`getClassNodes` `getAllDeclarations` `getRoutines` `getGlobalVariables` `getEnumerations`

Examples

```
fname = paste(tempfile(), ".c", sep = "")
cat("#include <stdio.h>\n", file = fname)
ll = system(paste("gcc -M", fname), intern = TRUE)
readDependencies(ll)

oname = tempfile()
ll = system(paste("gcc -M", fname, ">", oname), intern = TRUE)
readDependencies(oname)
```

resolveType *Create high-level, aggregate of translation unit nodes*

Description

This function converts information in a translation unit node into a higher-level, aggregate description of the associated data structure, class, routine, variable declaration.

Usage

```
resolveType(node, nodes, classDefs = DefinitionContainer(nodes), ...)
```

Arguments

node	the translation unit node from which to compute the aggregate information.
nodes	the entire array of nodes returned from parseTU. This is used to jump to other nodes not directly linked to node and resolve auxillary types in this function
classDefs	This is intended to be a place where previously resolved types are stored and can be referenced without having to resolve them again each time they are encountered. As such, it is a library of previously resolved types. It is also used to store information about what nodes are currently being processed so that infinite loops are avoided such as those that arise when a structure has a field that is a reference to the same data type as being defined, e.g `struct _A { struct _A *next;}`
...	

Author(s)

Duncan Temple Lang <duncan@wald.ucdavis.edu>

References

http://www.omegahat.org/RGccTranslationUnit

RGCCTranslationUnit-internal *Undocumented internal objects for RGCCTranslationUnit*

Description

These are internal functions for the moment that are not explicitly unexported, but are accessible to ease development. Some of these will be made available to developers and people using the package to do further processing of code rather than just using the top-level functionality.

This is an illustration of the need for a "protected" export for use in other packages by their authors, not the users.

`TypeDefinition-class`

Classes representing C/C++ types in a translation unit description

Description

This is a collection of related classes that are used to describe the basic and derived types in C/C++ code.

Structures and unions in C/C++ are different types of aggregators that contain one or more fields. In this framework, we use a common base class `ContainerDefinition` to represent the information so that we can define methods for that virtual class to be inherited by both actual types of data structures.

The `PendingType` class is a little different than the others. It is used when processing a type which may have references to itself or some other types that are also being processed recursively. The simplest example is a structure which has a field that is a pointer to an instance of the same class. This occurs in a linked list, for example. Then when resolving the definitin of this structure, we would end up resolving the type of this field and so would end up in an infinite loop. The call to `resolveType` arranges to check the collection of already resolved classes, and if it finds a definition for the target node, it uses that. When it starts actually resolving a previously unprocessed type, it puts a `PendingType` definition in the catalog of definitions to avoid the potential recursive loop.

`C++ReferenceType` is used for typedefs and aliases where we have a type definition that refers to another type definition. In this sense, the object is a type definition but the type it represents is another type definition. This is not to be confused with the `RC++Reference` class which is used at run-time as the base class for a reference/pointer to a C++ instance.

Objects from the Class

Instances of these classes are typically created via `resolveType`. Mostly, these are constructed directly using `new(class-name, ...)`.

Slots

`name:` Object of class `"character"` giving the "human" readable name for the type that is used when generating code.

`typeName:` Object of class `"character"`, identifying the base type of the pointer

`depth` an integer indicating the level of indirection of this pointer type, i.e. that this is a pointer to a pointer to a pointer ... A simple pointer to an actual type has a depth value of 1.

`qualifiers` a character vector returned from a call to `node$quals()` on the underlying Perl GCC::Node object. This gives the qualifiers for the node including things such as const, C, volatile, etc. that give us more information about the definition of the type.

`scope` currently, a character vector with zero or a single named entry. If this is the empty vector, then the scope is the top-level. If this contains an element, that value is the index of the node in the TU array/parser that defines the scope. The name of the element gives the human readable name. This field is used to determine whether the type is visible at the top-level, i.e. C/C++ code that interfaces to the code.

It is not clear that all TypeDefinition objects need this field, but it seems reasonable at the moment.

Methods

No methods defined with class "TypeDefinition" in the signature.

Author(s)

Duncan Temple Lang <duncan@wald.ucdavis.edu>

References

http://www.omegahat.org/RGCCTranslationUnit

See Also

`resolveType` The built-in primitive types i.e. `boolType-class`, etc.

`typeMap` *Create table of type mappings for generating interfaces*

Description

The code generation mechanism in this package allow for various types of customization. A type map is a table that identifies a native type (a target) and optional operations for converting between that type and R values. This function collects the arguments into a map. Each element applies to a particular native type, e.g. an int, an unsigned long, a pointer to a structure, a typedef. Each element is a list. The native type is identified in one of 2 ways. The name of the map element can be the familiar stringified declaration form of the native type, e.g. `int`, `FILE *`. Alternatively, the element can have a `target` element which is either a string in the same form as above, or an actual type object of class `TypeDefinition`.

Having identified the target native type, an element in this map can have any of the following elements coerceRValue, convertRValue, convertValueToR. Each of these can be a string giving the name of an R function or C routine as appropriate. The generated code will be a simple call to this function/routine with the parameter in the code. That is, if we are trying to convert a C object held in the variable `ans` back to R, and the convertValueToR element is given as `"R_matrix"`, the generated code is `R_matrix(ans)`.

If simply pasting the name of a function/routine and the variable name together is too simple, one can provide a function as the value of any of these elements. In that case, the function is called with three arguments: the name of the variable being processed, a list of all the variables being processed giving their names and types, and this type map which can be used for further conversions.

A function can return a simple string and the different functions use this in their default manner. Alternatively, one can return a string which is an RCode object and it will be inserted as-is with no subsequent processing, e.g. prefixed with `"var = "`.

Usage

```
typeMap(...)
```

Arguments

...	a collection of map elements. Names can be specified in the call and these can be used to match against the different type objects being looked up in the map. Each element can have elements named target, coerceRValue, convertRValue and convertValueToR. These can be function/routine names or functions that take 3 arguments.

Value

A list with S3 class `TypeMap` to help identify it in computations..

Author(s)

Duncan Temple Lang

References

~put references to the literature/web site here ~

See Also

`createMethodBinding` `coerceRValue` `convertRValue` `convertValueToR`

Examples

```
typeMap( "FILE *" = list(target = "FILE *",
                           # Can be a string, e.g. asFILE, but then couldn't get the mo
                         coerceRValue = function(name, ...) paste("asFILE(", name, ", '
                         convertRValue = "R_openFile",
                         convertValueToR = NULL
                        ))

 # coercion in R is a function and returns an as-is RCode block.
typeMap( "FILE *" = list(target = "FILE *",
                         coerceRValue = function(name, ...) {
                                          structure(paste("f = as(", name, ", 'FILERef'
                                                    class = "RCode")
                                        }
                        ))
```

---

writeCode *Output code for the bindings*

Description

The methods for `writeCode` are responsible for writing the contents of the generated code to a connection, or by default the console. After we generate code based on high-level descriptions of native routines, data structres, classes and methods, these methods take care of writing the different pieces to different files. We specify the generated code along with the target code type, i.e. native or R, to output the relevant pieces of the code. For example, we might write all the C code.

`writeIncludes` is a convenience function for adding a sequence of `#include file` directives to a connection. It can be instructed to put quotes or <> around each file name, or not!

Usage

```
writeCode(obj, target, file = stdout(), ..., includes = character())
```

Arguments

obj	the generated interface in different forms, for C++ classes, collection of methods, individual method interfaces, etc.
target	the target language, "r" or "native" at present. This indicates the language/context for which we are generating the code. In the future, we will add "header" for .h files to provide declarations that are used across files, and "namespace" for declarations that appear in a package's NAMESPACE file.
...	additional arguments for cat
file	the connection used when cat'ing the generated code.
includes	a character vector giving the file names (with relevant quotes or <> affixes) that will be included via the C/C++ preprocessor when the native code is compiled. Tis is a convenient way to put calls to include files in the code without explicitl calling writeIncludes having previously opened the connection. This does it after the connection is opened for you if you pass a file name for the connection.

Value

The return value is of no interest. The side-effect is important to produce the content on a connection.

Author(s)

Duncan Temple Lang <duncan@wald.ucdavis.edu>

References

http://www.omegahat.org/RGCCTranslationUnit

See Also

createClassBindings createMethodBinding createInterface

Examples

```
library(RGCCTranslationUnit)

 # Read the translation unit file for source code that defines several files.
my = parseTU(system.file("examples", "ABC.cpp.tu", package = "RGCCTranslationUnit"))
k = getClassNodes(my)

 # Generate the bindings for the class named B,
 # creating R and C++ code to interface to the class' methods.
z = createClassBindings(k$B, my)

# The C++ code
writeCode(z, "native")
```

```
 # The R code,
 writeCode(z, "r")

## Not run:
 library(RGCCTranslationUnit)
 tuFile = "inst/examples/ABC.cpp.tu"
 my = parseTU(tuFile)
 k = getClassNodes(my, "ABC")

  # Generate the bindings for the class named B,
  # creating R and C++ code to interface to the class' methods.
 aa = createClassBindings( k$A , my)
 bb = createClassBindings( k$B , my)
 cc = createClassBindings( k$C , my)

 f = file("inst/examples/RmyClassB.cpp"); open(f, "w")
 cat('#include "RConverters.h"', "\n", file = f)
 cat('\#include "ABC.h"', "\n\n\n", file = f)
# cat('extern "C" {', "\n\n\n", file = f)

 writeCode(aa, "native", file = f)
 writeCode(bb, "native", file = f)
 writeCode(cc, "native", file = f)

# cat('}', "\n", file = f)
 close(f)

 f = file("inst/examples/RABC.R"); open(f, "w")
 writeCode(aa, "r", file = f)
 writeCode(bb, "r", file = f)
 writeCode(cc, "r", file = f)
 close(f)

 writeCode(aa, "NAMESPACE")
 writeCode(bb, "NAMESPACE")
## End(Not run)
```

Index