SPOOR

Simple, Poor-man's Object-Oriented Run-time
Object-oriented extensions for C
Manual version $Revision: 2.18 $
$Date: 1994/11/29 02:04:29 $

Bob Glickstein

This manual documents SPOOR, a library package for C programs that provides object-oriented extensions.

Introduction

This manual describes Spoor, the Simple, Poor-man's Object-Oriented Run-time, which is a library package for C programs that provides object-oriented extensions to the language.

1 Overview

Spoor is a set of functions, macros, variables, types, and programming conventions which provide a totally portable object-oriented programming environment with polymorphism and single inheritance of data and methods.

A discussion of object-oriented programming is beyond the scope of this manual, but this chapter discusses the broad concepts in Spoor.

A *class* is a data type consisting of a structure with a set of fields and a set of *methods*. The class's methods are functions that operate on *instances* of the class. Methods are distinguished from ordinary functions because of inheritance and polymorphism.

1.1 Inheritance

Every Spoor class is a *subclass* of some other class, forming a hierarchy of *inheritance*. (One class, the *root class*, does not inherit from any other class; see Section 5.1 [The root class], page 13.) Inheritance means that the data structure contains all the fields of the superclass plus whatever the new class adds (*inheritance of data*); and the class has all the methods of the superclass plus whatever the new class adds (*inheritance of methods*). Furthermore, an inherited method may be *overridden*, replacing the inherited function with a new one.

1.2 Polymorphism

Polymorphism refers to the fact that an instance of a class can be treated as an instance of its superclass. Method invocation works properly on an instance even when it's being handled polymorphically. For instance, suppose class *foo* is a subclass of class *bar*, and that *f* is an instance of *foo*. Now suppose *f* is (polymorphically) passed to a function that expects an instance of *bar*. If that function now invokes a *bar* method on *f* and *foo* overrides that method, then the override in class *foo* will (correctly) be called, even though the caller doesn't know that *f* is an instance of *foo*.

Unfortunately, the C language has no concept of polymorphism. Therefore, if you pass a `struct foo *` where a `struct bar *` is expected, a strict type-checking C compiler will complain about a type mismatch, even though you and I know that class *foo* inherits from *bar*. The solution to this problem is to typecast the pointer before passing it, or to declare the recipient as taking a `void *` rather than a `struct bar *`. Both solutions throw away useful type information; this is Spoor's greatest drawback.

2 Fundamentals

A call to `spoor_Initialize` must precede any other SPOOR function calls.

`void spoor_Initialize ()` [Function]

Initialize SPOOR.

This function may raise the `strerror(ENOMEM)` exception.

After the first call to `spoor_Initialize`, subsequent calls are no-ops.

All SPOOR classes ultimately inherit from the root class, which is named `spoor` (see Section 5.1 [The root class], page 13). Because of polymorphism, any instance of any class may be referred to as a `struct spoor *`, and that is how this document indicates a function or some other context where an instance of any class is expected.

2.1 Class name

Every SPOOR class has a name, and by convention it is used in the following places:

- It is the "struct tag" for the data structure corresponding to the class (see Section 3.2 [The data structure], page 6);
- It is the human-readable *name* passed to `spoor_CreateClass` (see Section 4.5.1 [Creating the class descriptor], page 10);
- It is the prefix of accessor macro names for the class (see Section 3.3 [Accessors], page 6);
- It is the middle part of method selector variable names (see Section 2.3 [Invoking a method], page 4);
- It is the prefix of the new-instance-obtaining macro (see Section 3.5 [Other declarations], page 7);
- It is the prefix of the class descriptor variable name (see Section 4.2 [The class descriptor], page 8);
- It is the prefix of the class-initializing function name (see Section 4.5 [Class initializer], page 10);
- It is also helpful if the header file containing declarations for a class named *foo* is named `foo.h`, allowing modules that use that class to anticipate the name needed for the `#include` directive.

Much of the workings of SPOOR depends on following these conventions.

2.2 Obtaining an instance

A new instance can be allocated (via `malloc`) using `spoor_NewInstance`.

`struct spoor * spoor_NewInstance` (*struct spClass* *`class`) [Macro]

Allocate, initialize, and return a new instance of *class*. Allocation is performed with `malloc`; initialization is performed with `spoor_InitializeInstance` (q.v.).

This function may raise the `strerror(ENOMEM)` exception.

By convention, each class supplies a macro for allocating instances of itself. In class *foo*, for instance, the macro *foo*`_NEW()` is equivalent to `spoor_NewInstance(`*foo*`_class)`.

`void spoor_DestroyInstance` (*struct spoor *`obj`*) [Function]

Finalize and deallocate *obj*, which is an instance previously obtained with `spoor_NewInstance`. Finalizing is performed with `spoor_FinalizeInstance` (q.v.); deallocation is performed with `free`.

It is not necessary to rely on SPOOR to allocate and deallocate individual instances. The caller may provide storage for instances itself; in which case it is necessary to initialize and finalize the storage using `spoor_InitializeInstance` and `spoor_FinalizeInstance`.

`void spoor_InitializeInstance` (*struct spClass *`class`*, *struct spoor *`obj`*) [Macro]

Initialize *obj* as an instance of class *class*. Recursively calls the constructors for the superclasses of *class*, then calls the constructor for *class*, if one exists. See Section 4.3 [Constructor and destructor], page 8.

`void spoor_FinalizeInstance` (*struct spoor *`obj`*) [Macro]

Finalize *obj*, which is an instance that was previously initialized with `spoor_InitializeInstance`. Calls the destructor (if one exists) for the class to which *obj* belongs; then recursively calls the destructors for the superclasses of *obj*.

2.3 Invoking a method

Every method in a particular class has a *selector*. A selector is an integer which is used in the method table lookup. By convention, the selector for each method is held in a global variable of type `int` whose name is `m_foo_fum`, where *foo* is the name of the class which defines the method and *fum* is the name of the method.

If class *foo* inherits method *gronk* from its superclass, *bar*, then the selector will be in `m_bar_gronk`—even if *foo* overrides *gronk* with its own version. The class name in the method selector variable is the class that *adds* the method (via `spoor_AddMethod`), not any class that inherits or overrides the method.

To invoke a method on a particular instance, you *send* it the appropriate selector, plus any additional arguments that are required, with `spSend`.

`void spSend` (*struct spoor *`obj`*, *int `selector`*, ...) [Function]

Invoke the method on *obj* whose selector is *selector*. Remaining arguments, if any, are passed to the method.

The method-selecting function `spSend` can only be used to invoke `void`-returning methods, because it is of type `void` itself. A family of `spSend` variants exists, with one version for each possible scalar return type. All are called in the same way.

`spSend_c` Returns `char`.

`spSend_d` Returns `double`.

`spSend_f` Returns `float`.

`spSend_i` Returns `int`.

`spSend_l` Returns `long`.

`spSend_p` Returns `void *`.

`spSend_s` Returns `short`.

`spSend_uc`
: Returns `unsigned char`.

`spSend_ui`
: Returns `unsigned int`.

`spSend_ul`
: Returns `unsigned long`.

`spSend_us`
: Returns `unsigned short`.

3 The .h file

This chapter describes how to construct the `.h` file when defining a new SPOOR class. It is intended to be used as a "recipe." If you do the steps described in each of the following sections, you should end up with a complete SPOOR `.h` file.

3.1 The .h prelude

First, it's wise to wrap the entire `.h` file in a *condom* to protect it against multiple inclusions:

```
#ifndef FOO_H
# define FOO_H
...everything in the file...
#endif /* FOO_H */
```

Inside the condom, the file should first `#include <spoor.h>`, then `#include` the `.h` file for this class's superclass.

3.2 The data structure

The data structure for a SPOOR class must be a `struct` and must begin like this:

```
struct foo {
    SUPERCLASS(bar);
    ...
```

The `SUPERCLASS` macro inserts inheritance information into the data structure; its argument is the name of the superclass.[1]

The remainder of the `struct` definition may contain anything at all.

3.3 Accessors

Suppose class *foo* inherits from class *bar*. Suppose that instances of *bar* have a field named *b*. Unfortunately, it is not possible to write:

```
struct foo *f;

...
x = f->b;
...
```

because `struct foo` does not have a field named *b*, and the C compiler doesn't know to use the inherited field.

To get around this problem in SPOOR, there is a convention of writing *accessors* for each field in a class's data structure. An accessor is a macro that takes a pointer to an instance, casts it to the appropriate type, then dereferences a field. Example:

[1] In fact, the `SUPERCLASS` macro inserts an entire instance of the superclass, which in turn contains an instance of *its* superclass, and so on. In this way, a pointer to a `struct foo` can be cast to a pointer to a `struct bar` and a valid `struct bar` will be found at that address!

```
struct bar {
    SUPERCLASS(...);
    ...
    int b;
    ...
};

#define bar_b(x) (((struct bar *) (x))->b)
```

With this accessor, it is now possible to rewrite the earlier example as:

```
struct foo *f;

...
x = bar_b(f);
...
```

Notice that the accessor can be used as an lvalue[2] as well as a value. By convention, the name for an accessor of field *f* in class *c* is `c_f`.

Accessors may be a workaround for an annoying problem, but they do have one serendipitous benefit: *not* providing an accessor for a certain field is akin to making that field "private" in the C++ sense.

3.4 Declaring selectors

The `.h` file should declare the global variables in which method selectors will be held (see Section 2.3 [Invoking a method], page 4). For instance, if class *foo* has methods *a*, *b*, and *c*, the `.h` file should contain:

```
extern int m_foo_a;
extern int m_foo_b;
extern int m_foo_c;
```

Only methods newly added in this class need to have new selectors declared. Declarations for selectors for inherited and overridden methods are obtained by including the superclass's `.h` file.

3.5 Other declarations

The global class descriptor for the class needs to be declared. For a class named *foo*, it should be named `foo_class`, and is of type `struct spClass *`:

```
extern struct spClass *foo_class;
```

The class initializer needs to be declared. For a class named *foo*, it should be named `foo_InitializeClass`:

```
extern void foo_InitializeClass();
```

A macro for allocating instances of the class should be defined. For a class named *foo*, it should be named `foo_NEW`, and should look like this:

```
#define foo_NEW() \
    ((struct foo *) spoor_NewInstance(foo_class))
```

[2] An *lvalue* is something that can be on the left-hand side of an assignment operator.

4 The .c file

This chapter describes how to construct the `.c` file when defining a new SPOOR class. It is intended to be used as a "recipe." If you do the steps described in each of the following sections, you should end up with a complete SPOOR `.c` file.

4.1 The .c prelude

The `.c` file should first include its corresponding `.h` file.

4.2 The class descriptor

First, the global class descriptor variable should be defined and initialized to 0. For a class named *foo*, it should be named *foo*`_class`:

```
struct spClass *foo_class = 0;
```

Initializing *foo*`_class` to 0 allows other modules to test whether this one has been initialized yet; this is important in the class initializer (see Section 4.5 [Class initializer], page 10).

4.3 Constructor and destructor

Next, you may wish to define a constructor function and/or a destructor function. These should be declared `static`. The addresses of these functions will be passed to `spoor_CreateClass` in the class initializer (see Section 4.5 [Class initializer], page 10).

The constructor/destructor functions are both `void` and both take a single argument, an instance of the class. The constructor may assume that the superclass portion of the instance is already initialized, and should initialize the portion of the instance specific to the current class. The destructor should finalize the portion of the instance specific to the current class.

As an example, suppose class *foo* has the following data structure:

```
struct foo {
    SUPERCLASS(bar);
    struct dynstr d;
};
```

Then the constructor and destructor might look like this:

```
static void
foo_initialize(self)
    struct foo *self;
{
    dynstr_Init(&(self->d));
}
```

```
static void
foo_finalize(self)
    struct foo *self;
{
    dynstr_Destroy(&(self->d));
}
```

4.4 Methods

Each new method and override is its own function in the `.c` file. Each such function should be declared `static`. The addresses of these functions will be passed to `spoor_AddMethod` or `spoor_AddOverride` in the class initializer (see Section 4.5 [Class initializer], page 10).

Each function should be declared to take two arguments. The first argument is an instance of the current class. The second argument is of type `spArgList_t` and is an encapsulation of the method's "real" arguments.

4.4.1 Unpacking arguments

The first thing a method implementation should do is to *unpack* its arguments. Arguments to SPOOR methods are packaged up in an object of type `spArgList_t` and must be extracted in a varargs-like way. Unlike varargs, no initialization or finalizing of the argument-list object is required, and only a single pass over the arguments is permitted. The macro for extracting the next argument from the argument list object is called `spArg`.

`spArg` (*spArgList_t* ***arglist***, ***type***) [Macro]

Extract from the argument list object *arglist* the next argument, which is of type *type*. This macro is used exactly like `va_arg`.

Therefore, a method that takes an `int` and a string as arguments and returns a string should begin something like this:

```
static char *
foo_SomeMethod(self, arg)
    struct foo *self;
    spArgList_t arg;
{
    int i = spArg(arg, int);
    char *str = spArg(arg, char *);
    ...
```

4.4.2 Super calls

In the implementation of a method which overrides an inherited version of itself, it is possible to invoke the overridden version using `spSuper`. A *super call* looks like a normal method invocation, except that the class descriptor is included as an argument.

`void spSuper` (*struct spClass* ****class***, *struct spoor* ****obj***, *int* ***selector***, . . .) [Function]

Invoke the implementation of method *selector* which is hidden by the override in class *class*, acting on *obj*. Remaining arguments, if any, are passed to the method.

Like `spSend`, there are different versions of `spSuper` to call based on the return type of the method being invoked.

`spSuper_c`
: Returns `char`.

`spSuper_d`
: Returns `double`.

`spSuper_f`
: Returns `float`.

`spSuper_i`
: Returns `int`.

`spSuper_l`
: Returns `long`.

`spSuper_p`
: Returns `void *`.

`spSuper_s`
: Returns `short`.

`spSuper_uc`
: Returns `unsigned char`.

`spSuper_ui`
: Returns `unsigned int`.

`spSuper_ul`
: Returns `unsigned long`.

`spSuper_us`
: Returns `unsigned short`.

4.5 Class initializer

The class initializer is a `void` function of no arguments which sets up the run-time information about the class. It creates the class descriptor, linking it to its superclass. It fills in the class's method table with inherited methods, overrides, and new methods. In other words, it does all the things that a real object-oriented language does as compile-time.

The name of the class initializer for a class *foo* should be `foo_InitializeClass`.

4.5.1 Creating the class descriptor

First, the class initializer should make sure that the superclass is already fully initialized. Suppose class *foo* inherits from class *bar*:

```
if (!bar_class)
    bar_InitializeClass();
```

This works because of the convention of assigning `NULL` to the class descriptor before it's initialized.

Next, the class initializer should abort if its class is already initialized:

```
if (foo_class)
    return;
```

Next, the class descriptor should be created using `spoor_CreateClass`.

`struct spClass * spoor_CreateClass` (*char* ***name**, *char* ***doc**, *struct spClass* ***super**, *int* **size**, *void* (***init**)(*struct spoor* *), *void* (***final**)(*struct spoor* *)) [Function]

Create a new class named *name*. A brief, human-readable description of the class is in *doc*.[1] The class inherits from *super*. The size of an instance of the class is *size*. The functions *init* and *final* are a constructor and a destructor, respectively, for instances of the class. All of *doc*, *init*, and *final* may be `NULL`.

This function may raise the `strerror(ENOMEM)` exception.

To continue the example:

```
foo_class = spoor_CreateClass("foo", "a fooish class",
    bar_class, sizeof (struct foo),
    foo_initialize, foo_finalize);
```

4.5.2 Overriding inherited methods

The class initializer should next call `spoor_AddOverride` on any methods that the class has inherited and wishes to override.

`void spoor_AddOverride` (*struct spClass* ***class**, *int* **selector**, *char* ***doc**, *spoor_method_t* **fn**) [Macro]

Override the method inherited by *class* whose selector is *selector*, using *doc* as the new human-readable documentation string and *fn* as the new implementation. The type of *fn*, which is `spoor_method_t`, is "function returning any type, taking a Spoor instance and a `spArgList_t` as arguments." If *doc* is `NULL`, then the inherited documentation string is used. The function *fn* may return any scalar type. If *fn* is `NULL`, then the method is overridden to become an abstract method (see Section 4.5.3 [Adding new methods], page 11).

4.5.3 Adding new methods

The class initializer should next call `spoor_AddMethod` to add any new methods and to initialize the method selector variables.

`int spoor_AddMethod` (*struct spClass* ***class**, *char* ***name**, *char* ***doc**, *spoor_method_t* **fn**) [Macro]

Add a new method to *class* named *name*, with human-readable documentation string *doc* and implementation *fn*. Return value is the selector for this method, which should be assigned to a global variable whose name is `m_foo_name`, for a class named *foo*.

If *fn* is `NULL`, then the newly-added method is called an *abstract method*. An abstract method is one which has no implementation in a particular class but which is generally expected to be overridden with real implementations in subclasses.

[1] The *name* and *doc* parameters are included on the off chance that someday, someone will want a Spoor class browser. They also aid in debugging.

Invoking a method on an object in whose class that method is abstract causes the invocation on the same object of the method `subclassResponsibility`, which is inherited from the root class (see Section 5.1 [The root class], page 13). The default action of this method is to raise the exception `spoor_SubclassResponsibility`. Naturally, that method may be overridden.

This function may raise the `strerror(ENOMEM)` exception.

4.5.4 Other class initializer code

After all new methods have been added, the class initializer should call the class initializers of any other classes on which it depends. For instance, if class *foo* uses class *other* somewhere in its implementation, it should call `other_InitializeClass()` after *foo* is fully initialized. This may result in many calls to `other_InitializeClass` as all the dependent classes call it; but if it follows the conventions set forth here, any but the first call should be an inexpensive no-op.

Finally, the class initializer should initialize any class-specific data; that is, any data shared by all instances of the class. For instance, a text widget class would initialize a clipboard buffer shared by all text widgets.

5 Miscellaneous

This chapter describes other facilities of Spoor.

5.1 The root class

The root class is the predefined Spoor class from which all other classes ultimately derive. Its name is `spoor`, hence pointers to instances are of type `struct spoor *`. The class descriptor for the `spoor` class is in the global variable `spoor_class`. The `spoor` class provides two methods for all other classes to inherit.

`void subclassResponsibility` (*char *`class`*, *char *`method`*) [Method on `spoor`]

Automatically invoked by `spSend` and `spSuper` (and variously-typed variants) when attempting to invoke an abstract method (see Section 4.5.3 [Adding new methods], page 11). Arguments are *class*, the name of the class from which the error arose; and *method*, the name of the abstract method whose invocation was attempted. The action of the default version of this method is to raise the exception `spoor_SubclassResponsibility`, with the "exception value" set to the string `"class:method"` (suitable for retrieval with `except_GetExceptionValue`).

The selector for this method is in the global variable `m_spoor_subclassResponsibility`.

`void setInstanceName` (*char *`name`*) [Method on `spoor`]

Sets the name of an object to *name*. Spoor maintains a registry of named objects and allows object lookup by name using `spoor_FindInstance` (see Section 5.2 [Auxiliary functions], page 14). If *name* is `NULL`, then any name associated with the object is removed and the object is removed from the registry (if it was in it).

The variable `spoor_InstanceRegistry` is of type `struct hashtab *` and contains the registry. Callers may wish to iterate over its contents using `hashtab_Iterate` or obtain its size using `hashtab_Length`. Elements in this hash table have type `struct spoor_RegistryElt`, which is defined in `spoor.h` like so:

```
struct spoor_RegistryElt {
    char *name;
    struct spoor *obj;
};
```

The selector for this method is in the global variable `m_spoor_setInstanceName`.

This function may raise the `strerror(ENOMEM)` exception.

The `spoor` class also provides these accessors:

`struct spClass * spoor_Class` (*struct spoor *`obj`*) [Accessor]

Accessor for the class to which *obj* belongs. This accessor should never be assigned to!

`char * spoor_InstanceName` (*struct spoor *`obj`*) [Accessor]

Accessor for the name of *obj* as set by `setInstanceName`. This accessor should not be assigned to directly, since `setInstanceName` performs the additional step of placing the object in the named-instance registry.

Finally, the `spoor` class defines a macro for allocating new instances of itself, per convention:

`struct spoor * spoor_NEW` () [Macro]

Allocate, initialize, and return a new instance of the `spoor` class. Equivalent to calling `spoor_NewInstance(spoor_class)`.

This function may raise the `strerror(ENOMEM)` exception.

5.2 Auxiliary functions

`int spoor_IsClassMember` (*struct spoor *`obj`*, *struct spClass *`class`*) [Function]

Returns non-zero if *obj* is a member of *class* or some class that inherits from *class*; zero otherwise. This function calls `spoor_NumberClasses` (q.v.) if classes have not yet been numbered or if the creation of a new class has invalidated an old class numbering. The class numbering scheme makes `spoor_IsClassMember` a fast constant-time operation.

`void spoor_NumberClasses` () [Function]

Numbers all existing Spoor classes according to a depth-first traversal of the inheritance hierarchy. This operation is linear in the number of classes that exist, but it makes `spoor_IsClassMember` a nearly-instantaneous operation. In fact, `spoor_IsClassMember` will call `spoor_NumberClasses` automatically if necessary; however, the caller may wish to call this function itself if it wants precise control over when the O(n) overhead occurs.

`struct spoor * spoor_FindInstance` (*char *`name`*) [Function]

Find the instance named *name* and return it, or `NULL` if no such instance was found. The instance must have been named using the `setInstanceName` method (see Section 5.1 [The root class], page 13). If two or more instances have the same name, there is no way to know which one will be returned from this function.

`void spClass_setup` (*struct spClass *`class`*, *char *`name`*, *char *`doc`*, *struct spClass *`super`*, *int `size`*, *void (*`init`)(struct spoor *)*, *void (*`final`)(struct spoor *)*) [Function]

Like `spoor_CreateClass` (q.v.), but used when initializing a class descriptor, *class*, already obtained separately. In fact, `spoor_CreateClass` calls this function to initialize the class after allocating it.

This function may raise the `strerror(ENOMEM)` exception.

`struct spClass * spoor_FindClass` (*char *`name`*) [Function]

Find the Spoor class named *name* and return its descriptor, or `NULL` if no such class was found.

`int spoor_FindMethod` (*struct spClass *`class`*, *char *`name`*) [Macro]

Find the method in *class* named *name* and return its selector. The method may be one added or inherited by *class*. Return value is less than zero if no such method was found.

`char * spoor_MethodDescription` (*struct spClass* `*class`, *int* `selector`) [Macro]

Returns the documentation string (as given to `spoor_AddMethod` or `spoor_AddOverride`) for the method in *class* whose selector is *selector*.

5.3 The class class

The type of SPOOR class descriptors, `spClass`, is itself a SPOOR class! In fact, many SPOOR operations are actually implemented as methods in the `spClass` class.

The class descriptor for the `spClass` class is in the global variable `spClass_class`. The superclass for `spClass` is `spoor`. An allocator, `spClass_NEW()`, is defined.

No instance of `spClass` should ever be finalized or deallocated.

The macros `spoor_AddMethod`, `spoor_AddOverride`, `spoor_FindMethod`, `spoor_MethodDescription`, `spoor_InitializeInstance`, and `spoor_NewInstance` are all implemented in terms of methods on the `spClass` class. The corresponding selectors are `m_spClass_addMethod`, `m_spClass_addOverride`, `m_spClass_findMethod`, `m_spClass_methodDescription`, `m_spClass_initializeInstance`, and `m_spClass_newInstance`.

The `spClass` class defines these accessors:

`char * spClass_Name` (*struct spClass* `*class`) [Accessor]

Accessor for the name of the class described by *class*. This accessor should not be assigned to.

`struct spClass * spClass_superClass` (*struct spClass* `*class`) [Accessor]

Accessor for the descriptor of the superclass for the class described by *class*. If *class* is `spoor_class`, the result will be `NULL`. This accessor should not be assigned to.

6 Subsystems

This chapter describes the subsystems used by Spoor.

6.1 Except

Except is a portable package of functions and macros permitting an exception-handling programming style. Functions in Spoor which fail do not return error codes, they raise exceptions.

6.2 Dynadt

Dynadt is a portable library of several reusable, dynamically-resizing, container-style data types, including an array type (used in Spoor in the implementation of method tables) and a hash table (used in Spoor in the implementation of the named instance registry).

7 Shortcomings

Spoor has some known shortcomings.

weak type-checking

Object-oriented programming entails polymorphism, but C doesn't know what polymorphism is. As a result, it is necessary to typecast often or to use `void *` when you mean some more specific type. See Section 1.2 [Polymorphism], page 2.

Glossary

abstract method
: A *method* with no implementation. The purpose of such a method is to provide a common entrypoint for *subclasses* to *inherit*. The subclasses *override* the missing method implementation with implementations of their own.

accessor
: A macro for referring to the fields of a SPOOR class's data structure. Accessors are needed because the C compiler won't automatically cast pointers to allow you to refer to *inherited* fields.

argument unpacking
: See "unpacking."

class
: A data structure and a set of *methods*. The basis for the data structure and for the set of methods are *inherited* from the *superclass*.

condom
: A preprocessor-based convention for protecting a header file against multiple inclusions.

inheritance
: The copying of data fields and methods from a *superclass* to a *subclass*. The subclass usually augments or overrides the inherited behavior.

instance
: An object belonging to some *class*. "Belonging" in this case means that the object's type is that of the class's data structure, and that the class's methods can operate on the object.

instance name
: See "named instance."

method
: A function belonging to a *class* and *inherited* (and possibly *overridden*) by *subclasses*. Each method operates on an *instance* of the class and any needed additional arguments.

method selector
: See "selector."

named instance
: An *instance* which has had a name associated with it via the *root class method* `setInstanceName`. Such instances can be retrieved by name from a global instance *registry*.

override
: To replace the *inherited* implementation of a *method* with a different function.

polymorphism
: The property that allows *instances* of some *class* to be handled as instances of the *superclass* but still behave like instances of the correct class (for purposes including *method* invocation).

registry
: A global table of *named instances* which can be accessed with the function `spoor_FindInstance`.

root class
: The *class* which does not *inherit* from any other and forms the root of the inheritance hierarchy.

selector
: An integer representing a *method*. Used in calls to `spSend` and `spSuper` to select the method to invoke on an *instance*.

send
: To invoke a *method* on an *instance*. The method *selector* is said to be "sent" to the instance.

subclass
: A *class* which *inherits* from another.

subclass responsibility
: The error in attempting to invoke an *abstract method* (rather than a concrete *override* in some *subclass*). It is the subclass's responsibility to provide an implementation for the requested method.

super call
: In an *override*, a call (using `spSuper`) to the overridden version of the method.

superclass
: A *class* from which another *inherits*.

unpacking
: The process of extracting arguments to a Spoor *method* one at a time using a varargs-like macro, `spArg`.

virtual function
: See "abstract method".

Index

A

abstract method ... 11
accessor ... 6
addMethod ... 15
addOverride ... 15
argument unpacking ... 9

C

class ... 2
condom ... 6

F

findMethod ... 15

I

inheritance ... 2
initializeInstance ... 15
instance ... 2
instance name ... 13

M

m_spClass_addMethod ... 15
m_spClass_addOverride ... 15
m_spClass_findMethod ... 15
m_spClass_initializeInstance ... 15
m_spClass_methodDescription ... 15
m_spClass_newInstance ... 15
m_spoor_setInstanceName ... 13
m_spoor_subclassResponsibility ... 13
method ... 2
method selector ... 4
methodDescription ... 15

N

named instance ... 13
newInstance ... 15

O

override ... 2

P

polymorphism ... 2

R

registry ... 13
root class ... 2

S

selector ... 4
send ... 4
setInstanceName ... 13
spArg ... 9
spArgList_t ... 9
spClass ... 7
spClass_class ... 15
spClass_Name ... 15
spClass_NEW ... 15
spClass_setup ... 14
spClass_superClass ... 15
spoor ... 13
spoor_AddMethod ... 11
spoor_AddOverride ... 11
spoor_Class ... 13
spoor_class ... 13
spoor_CreateClass ... 11
spoor_DestroyInstance ... 4
spoor_FinalizeInstance ... 4
spoor_FindClass ... 14
spoor_FindInstance ... 14
spoor_FindMethod ... 14
spoor_Initialize ... 3
spoor_InitializeInstance ... 4
spoor_InstanceName ... 13
spoor_InstanceRegistry ... 13
spoor_IsClassMember ... 14
spoor_method_t ... 11
spoor_MethodDescription ... 15
spoor_NewInstance ... 3
spoor_NEW ... 14
spoor_NumberClasses ... 14
spoor_SubclassResponsibility ... 13
spSend ... 4
spSend_c ... 4
spSend_d ... 4
spSend_f ... 4
spSend_i ... 4
spSend_l ... 4
spSend_p ... 4
spSend_s ... 4
spSend_uc ... 5
spSend_ui ... 5
spSend_ul ... 5
spSend_us ... 5
spSuper ... 9
spSuper_c ... 10
spSuper_d ... 10
spSuper_f ... 10
spSuper_i ... 10
spSuper_l ... 10
spSuper_p ... 10
spSuper_s ... 10

spSuper_uc ... 10
spSuper_ui ... 10
spSuper_ul ... 10
spSuper_us ... 10
subclass ... 2
subclass responsibility ... 11
subclassResponsibility ... 13
super call ... 9

U

unpacking arguments ... 9

Table of Contents

Introduction **1**

1 Overview **2**
- 1.1 Inheritance 2
- 1.2 Polymorphism 2

2 Fundamentals **3**
- 2.1 Class name 3
- 2.2 Obtaining an instance 3
- 2.3 Invoking a method 4

3 The .h file **6**
- 3.1 The .h prelude 6
- 3.2 The data structure 6
- 3.3 Accessors 6
- 3.4 Declaring selectors 7
- 3.5 Other declarations 7

4 The .c file **8**
- 4.1 The .c prelude 8
- 4.2 The class descriptor 8
- 4.3 Constructor and destructor 8
- 4.4 Methods 9
  - 4.4.1 Unpacking arguments 9
  - 4.4.2 Super calls 9
- 4.5 Class initializer 10
  - 4.5.1 Creating the class descriptor 10
  - 4.5.2 Overriding inherited methods 11
  - 4.5.3 Adding new methods 11
  - 4.5.4 Other class initializer code 12

5 Miscellaneous **13**
- 5.1 The root class 13
- 5.2 Auxiliary functions 14
- 5.3 The class class 15

6 Subsystems **16**
- 6.1 Except 16
- 6.2 Dynadt 16

7 Shortcomings **17**

Glossary **18**

Index **20**