1 GNU Objective-C runtime features

This document is meant to describe some of the GNU Objective-C runtime features. It is not intended to teach you Objective-C, there are several resources on the Internet that present the language. Questions and comments about this document to Ovidiu Predescu <ovidiu@cup.hp.com>.

1.1 `+load`: Executing code before main

The GNU Objective-C runtime provides a way that allows you to execute code before the execution of the program enters the `main` function. The code is executed on a per-class and a per-category basis, through a special class method `+load`.

This facility is very useful if you want to initialize global variables which can be accessed by the program directly, without sending a message to the class first. The usual way to initialize global variables, in the `+initialize` method, might not be useful because `+initialize` is only called when the first message is sent to a class object, which in some cases could be too late.

Suppose for example you have a `FileStream` class that declares `Stdin`, `Stdout` and `Stderr` as global variables, like below:

```
FileStream *Stdin = nil;
FileStream *Stdout = nil;
FileStream *Stderr = nil;

@implementation FileStream

+ (void)initialize
{
    Stdin = [[FileStream new] initWithFd:0];
    Stdout = [[FileStream new] initWithFd:1];
    Stderr = [[FileStream new] initWithFd:2];
}

/* Other methods here */
@end
```

In this example, the initialization of `Stdin`, `Stdout` and `Stderr` in `+initialize` occurs too late. The programmer can send a message to one of these objects before the variables are actually initialized, thus sending messages to the `nil` object. The `+initialize` method which actually initializes the global variables is not invoked until the first message is sent to the class object. The solution would require these variables to be initialized just before entering `main`.

The correct solution of the above problem is to use the `+load` method instead of `+initialize`:

```
@implementation FileStream
```

```
+ (void)load
{
    Stdin = [[FileStream new] initWithFd:0];
    Stdout = [[FileStream new] initWithFd:1];
    Stderr = [[FileStream new] initWithFd:2];
}

/* Other methods here */
@end
```

The +load is a method that is not overridden by categories. If a class and a category of it both implement +load, both methods are invoked. This allows some additional initializations to be performed in a category.

This mechanism is not intended to be a replacement for +initialize. You should be aware of its limitations when you decide to use it instead of +initialize.

1.1.1 What you can and what you cannot do in +load

The +load implementation in the GNU runtime guarantees you the following things:

- you can write whatever C code you like;
- you can send messages to Objective-C constant strings (@"this is a constant string");
- you can allocate and send messages to objects whose class is implemented in the same file;
- the +load implementation of all super classes of a class are executed before the +load of that class is executed;
- the +load implementation of a class is executed before the +load implementation of any category.

In particular, the following things, even if they can work in a particular case, are not guaranteed:

- allocation of or sending messages to arbitrary objects;
- allocation of or sending messages to objects whose classes have a category implemented in the same file;

You should make no assumptions about receiving +load in sibling classes when you write +load of a class. The order in which sibling classes receive +load is not guaranteed.

The order in which +load and +initialize are called could be problematic if this matters. If you don't allocate objects inside +load, it is guaranteed that +load is called before +initialize. If you create an object inside +load the +initialize method of object's class is invoked even if +load was not invoked. Note if you explicitly call +load on a class, +initialize will be called first. To avoid possible problems try to implement only one of these methods.

The +load method is also invoked when a bundle is dynamically loaded into your running program. This happens automatically without any intervening operation from you. When you write bundles and you need to write +load you can safely create and send messages to objects whose classes already exist in the running program. The same restrictions as above apply to classes defined in bundle.

1.2 Type encoding

The Objective-C compiler generates type encodings for all the types. These type encodings are used at runtime to find out information about selectors and methods and about objects and classes.

The types are encoded in the following way:

`char`	`c`
`unsigned char`	`C`
`short`	`s`
`unsigned short`	`S`
`int`	`i`
`unsigned int`	`I`
`long`	`l`
`unsigned long`	`L`
`long long`	`q`
`unsigned long long`	`Q`
`float`	`f`
`double`	`d`
`void`	`v`
`id`	`@`
`Class`	`#`
`SEL`	`:`
`char*`	`*`
unknown type	`?`
bitfields	`b` followed by the starting position of the bitfield, the type of the bitfield and the size of the bitfield (the bitfields encoding was changed from the NeXT's compiler encoding, see below)

The encoding of bitfields has changed to allow bitfields to be properly handled by the runtime functions that compute sizes and alignments of types that contain bitfields. The previous encoding contained only the size of the bitfield. Using only this information it is not possible to reliably compute the size occupied by the bitfield. This is very important in the presence of the Boehm's garbage collector because the objects are allocated using the typed memory facility available in this collector. The typed memory allocation requires information about where the pointers are located inside the object.

The position in the bitfield is the position, counting in bits, of the bit closest to the beginning of the structure.

The non-atomic types are encoded as follows:

pointers	`'^'` followed by the pointed type.
arrays	`'['` followed by the number of elements in the array followed by the type of the elements followed by `']'`
structures	`'{'` followed by the name of the structure (or '?' if the structure is unnamed), the '=' sign, the type of the members and by `'}'`

unions	`'('` followed by the name of the structure (or '?' if the union is unnamed), the '=' sign, the type of the members followed by `')'`

Here are some types and their encodings, as they are generated by the compiler on a i386 machine:

Objective-C type	Compiler encoding
`int a[10];`	`[10i]`
`struct { int i; float f[3]; int a:3; int b:2; char c; }`	`{?=i[3f]b128i3b131i2c}`

```
struct {
  int i;
  float f[3];
  int a:3;
  int b:2;
  char c;
}
```

In addition to the types the compiler also encodes the type specifiers. The table below describes the encoding of the current Objective-C type specifiers:

Specifier	Encoding
`const`	`r`
`in`	`n`
`inout`	`N`
`out`	`o`
`bycopy`	`O`
`oneway`	`V`

The type specifiers are encoded just before the type. Unlike types however, the type specifiers are only encoded when they appear in method argument types.

1.3 Garbage Collection

Support for a new memory management policy has been added by using a powerful conservative garbage collector, known as the Boehm-Demers-Weiser conservative garbage collector. It is available from `http://reality.sgi.com/boehm_mti/gc.html`.

To enable the support for it you have to configure the compiler using an additional argument, *`--enable-objc-gc`*. You need to have garbage collector installed before building the compiler. This will build an additional runtime library which has several enhancements to support the garbage collector. The new library has a new name, *`libobjc_gc.a`* to not conflict with the non-garbage-collected library.

When the garbage collector is used, the objects are allocated using the so-called typed memory allocation mechanism available in the Boehm-Demers-Weiser collector. This mode requires precise information on where pointers are located inside objects. This information is computed once per class, immediately after the class has been initialized.

There is a new runtime function `class_ivar_set_gcinvisible()` which can be used to declare a so-called **weak pointer** reference. Such a pointer is basically hidden for the garbage collector; this can be useful in certain situations, especially when you want to keep track of the allocated objects, yet allow them to be collected. This kind of pointers can only be members of objects, you cannot declare a global pointer as a weak reference. Every type which is a pointer type can be declared a weak pointer, including `id`, `Class` and `SEL`.

Here is an example of how to use this feature. Suppose you want to implement a class whose instances hold a weak pointer reference; the following class does this:

```
@interface WeakPointer : Object
{
    const void* weakPointer;
}

- initWithPointer:(const void*)p;
- (const void*)weakPointer;
@end


@implementation WeakPointer

+ (void)initialize
{
  class_ivar_set_gcinvisible (self, "weakPointer", YES);
}

- initWithPointer:(const void*)p
{
  weakPointer = p;
  return self;
}

- (const void*)weakPointer
```

```
{
  return weakPointer;
}

@end
```

Weak pointers are supported through a new type character specifier represented by the '!' character. The `class_ivar_set_gcinvisible()` function adds or removes this specifier to the string type description of the instance variable named as argument.