Creating Packages in R

Once you have a set of R functions (and optionally datasets and C or Fortran source code), the `package.skeleton()` function makes it very easy to create an R package. You pass this function the name of a package you wish to create along with a vector containing the names of R objects in the current environment to be included in the package. A directory with the name of the program will be created (by default in the current directory) containing a skeleton `DESCRIPTION` file, along with `man`, `R`, and `src` subdirectories. Skeleton documentation files for each of the objects you passed to `package.skeleton()` can be found in the `man` subdirectory. In addition, a `README` file contains instructions on how to proceed.

Source Code and Dynamic Loading of Libraries

If you are using any C or Fortran code in your package, you need to copy the necessary files into the `src` directory. The package will run `R CMD SHLIB` on these files to create a shared object file of the same name as your package. To insure that the library is properly loaded, you should modify the file which has been placed in the `R` directory to include a `.First.lib()` function, which will be called whenever your library is loaded. A function like this one is usually sufficient:

```
".First.lib" <-
function(libname,pkgname)
        library.dynam("package",pkgname,libname)
```

where the first argument to `library.dynam()` is your package name in quotes without any suffix.

Filling in the Documentation Skeletons

For each function and dataset included in your package, there will be a skeleton `.Rd` file in the `man` directory. The skeleton file looks something like this:

```
\name{somefunction}
\alias{somefunction}
%- Also NEED an '\alias' for EACH other topic documented here.
\title{ ~~function to do ... ~~ }
\description{
  ~~ A concise (1-5 lines) description of what the function does. ~~
}
\usage{
somefunction(arg1, arg2, arg3, arg4 = 10)
}
%- maybe also 'usage' for other objects documented here.
\arguments{
  \item{arg1}{ ~~Describe \code{arg1} here~~ }
  \item{arg2}{ ~~Describe \code{arg2} here~~ }
  \item{arg3}{ ~~Describe \code{arg3} here~~ }
  \item{arg4}{ ~~Describe \code{arg4} here~~ }
}
\details{
  ~~ If necessary, more details than the __description__  above ~~
}
\value{
  ~Describe the value returned
  If it is a LIST, use
```

Filling in the Documentation Skeletons(cont'd)

```
  \item{comp1 }{Description of 'comp1'}
  \item{comp2 }{Description of 'comp2'}
  ...
}
\references{ ~put references to the literature/web site here ~ }
\author{ ~~who you are~~ }
\note{ ~~further notes~~ }

 ~Make other sections like Warning with \section{Warning }{....} ~

\seealso{ ~~objects to See Also as \code{\link{~~fun~~}}, ~~~ }
\examples{
##---- Should be DIRECTLY executable !! ----
##-- ==>  Define data, use random,
##--or do  help(data=index)  for the standard data sets.

\keyword{ ~kwd1 }% at least one, from doc/KEYWORDS
\keyword{ ~kwd2 }% __ONLY ONE__ keyword per line
```

Filling in the DESCRIPTION Skeleton

The initial `DESCRIPTION` file looks like this:

```
Package: packagename
Type: Package
Title: What the package does (short line)
Version: 1.0
Date: 2005-10-10
Author: Who wrote it
Maintainer: Who to complain to <yourfault@somewhere.net>
Description: More about what it does (maybe more than one line)
License: What license is it under?
```

Additional fields for `Depends`, `Suggests`, and `URL`, among others, can be added to this file. The building process will add a `Built` field, which needs to be removed before rebuilding the package.

Testing and building the package

If your R package doesn't contain any source code, you can proceed to building the package, but if it does contain source code you need to test to insure that the package builds properly. To test the build, use a command like

```
R CMD check packagename
```

Try to resolve as many errors and warnings as possible at this stage, although it may be difficult or impossible to remove them all. Finally, when you're ready to create the package, type

```
R CMD build packagename
```

This will result in a gzipped tar file in the current directory, containing the package. You can now test it, either on the same system you built it on, or, preferably, some other architecture.

Calling C programs from R

There are three steps to making a C routine available to R:

1. Properly building a shared object suitable for dynamic loading. This is most easily accomplished with the command

   ```
   R CMD SHLIB list-of-cfiles
   ```

2. Dynamically loading the shared object into a running version of R. This is done with `dyn.load()` for a non-packaged function, and with `library.dynam` (via `.First.lib()`) for a function in a package.

3. Accessing the function from within R using the `.C()` function.

Building the shared object

Usually the `SHLIB` command will create a suitable shared object, but sometimes additional flags need to be sent to the preprocessor (for example `-I` flags to get headers from other directories), the C compiler, or the loader (for example `-L` flags specifying additional directories to be searched for supporting libraries). Such options can be set by creating a file in the compilation directory called `Makevars` with definitions of any of the following variables:

`PKG_CPPFLAGS`	pre-processor flags
`PKG_CFLAGS`	C compiler flags
`PKG_LIBS`	loader flags

Dynamically Loading the shared object

A simple call of the form

```
dyn.load("./myobject.so")
```

before using your package should be sufficient for most simple cases. If you wish to automate the process, you can call the `is.loaded()` function inside of any function that requires the shared object, and do the loading only if necessary:

```
if(!is.loaded("somefunc"))
    dyn.load("./myobject.so")
```

Here, `somefunc` represents the name of any of the C functions contained in your shared object.

Invoking your function using `.C()`

The `.C()` takes as its first argument the name of one of the functions defined in your shared object, and additional arguments for each of the objects that is passed to the function through its argument list. The function also accepts arguments called `NAOK`, `DUP`, and `PACKAGE`.

All function calls through `.C()` are performed using call-by-address; this means that all of the arguments to your C function must be declared as pointers. Furthermore, the usual mechanism of returning a value through the function name will not work; all information must be passed through the argument list. Often this means writing a small wrapper function, as illustrated in the next slide.

Creating wrapper functions

Suppose you have a function with a prototype such as the following:

```
double dosomething(double *x, long n,double alpha);
```

To use the function from R, a wrapper similar to the following could be constructed:

```
void Rdosomething(double *x,long *n,double *alpha,double *result)
{
   double theresult;
   theresult = dosomething(x,*n,*alpha);
   *result = theresult;
}
```

Argument types in `.C()`

When the `.C()` function is called R manipulates its internal pointers so that they are suitable for use in the C environment, and it is critical that it knows the type of each of the arguments. The mapping of C types to R types is shown below; functions like `as.double` or `as.integer` can be used when passing arguments to `.C()` to make certain the correct types of arguments are passed.

C Type	R Type
`float*`	`single`
`double*`	`double`
`long*`	`integer`
`char**`	`character`

A Simple Example

Consider the following C program, which generates a vector of random numbers:

```
#include <stdlib.h>
#include <limits.h>
#include <time.h>

void uni(long *n,double *res)
{  time_t now;
   unsigned long a=800000003, c=78900934;
   unsigned long x;
   long i;

   x = time(&now);
   for(i=0;i<*n;i++){
      x = a * x  + c;
      res[i] = (double)x / ULONG_MAX;
   }
}
```

Note that both arguments to the function are pointers. If the program is stored in the file `uni.c`, the command

```
R CMD SHLIB uni.c
```

will create a shared object called `uni.so` in the current directory.

Calling the C routine

It's usually a good idea to write a function which will insure that the shared object is loaded and that any necessary memory is "allocated" in the R environment before the C function is called. For this example, here's such a function:

```
"myuni" = function(n){
   result = double(n)  # or rep(0,n)
   if(!is.loaded(symbol.C("uni")))
     dyn.load("./uni.so")

   z = .C("uni",as.integer(n),result=as.double(result))
   z$result
}
```

The `.C()` function returns a list containing the values of each of the arguments passed to your C function; note that by naming the components, desired results can be easily extracted. In this example, the result could also be accessed through `z[[2]]`.

Memory Allocation

In the previous example, the memory necessary to hold the result vector was "allocated" in the R environment, not the C environment. Since R manages this memory, this approach should be used whenever feasible.

If you do need to allocate memory from within your C routine, you have two options. You can use the normal C library memory allocation routines, but remember that there will be no automatic freeing of memory, so it's your responsibility to call `free()` appropriately.

The other option is to use internal functions provided by R, which will free any allocated memory when the execution of `.C()` is complete. The function prototypes for the available routines are:

```
char* R_alloc(long n, int size);  # memory is not zeroed
char* S_alloc(long n, int size);  # memory is zeroed
char* S_realloc(char *p,long new, long old, int size)  #like realloc(3)
```

Using Matrices with `.C()`

In R, matrices are stored as one-dimensional vectors, stacked by rows, and you must store matrices in the same way in your C program in order for the `.C()` interface to work. Consider the following function, which scales element of a matrix by the maximum value of its column:

```
void mscale(double *mat,long *nrow,long *ncol)
{
   long i,j;
   double *matnow;
   double themax;

   for(i=0;i<*ncol;i++){
        matnow = mat + i * *nrow;
        themax = *(matnow++);
        for(j=1;j<*nrow;j++,matnow++)
            if(*matnow > themax)
                themax = *matnow;
        matnow = mat + i * *nrow;
        for(j=0;j<*nrow;j++,matnow++)
                *matnow /= themax;

   }
}
```

Using Matrices with `.C()` (cont'd)

The following function can be used to access the C program:

```
scalebycol = function(x){
    nrow = nrow(x)
    ncol = ncol(x)
    if(!is.loaded(symbol.C("mscale")))dyn.load("./scalemat.so")
    z = .C("mscale",x=as.double(x),as.integer(nrow),as.integer(ncol))
    z$x
 }
```

A quick test reveals a small problem:

```
> scalebycol(matrix(rnorm(15),5,3))
 [1]  1.0000 -1.4739 -0.4714  0.4115 -0.2359  0.9684 -1.0833
 [8] -0.9865 -1.3479  1.0000 -0.8453 -1.0376  1.0000  0.8600
[15] -0.6168
```

After returning from `.C()`, the result is no longer a matrix.

Using Matrices with `.C()` (cont'd)

Since the elements of the matrix are stored in the correct order internally, we can simply return `matrix(z$x,nrow,ncol)` instead of `z$x`.

An alternative to calling `matrix` on the returned value is to avoid the use of `as.double` and specify the storage mode of the matrix directly, before the call to `.C()`:

```
scalebycol = function(x){
    nrow = nrow(x)
    ncol = ncol(x)

    if(!is.loaded(symbol.C("mscale")))dyn.load("./scalemat.so")

    storage.mode(x) = "double"
    z = .C("mscale",x=x,as.integer(nrow),as.integer(ncol))
    z$x
 }
```