The .Call Interface I

- the R side of the .Call interface is almost the same as the .C interface
- lets recap the what we did in R for .C (remember, our C file was called prog1.c):

```
dyn.load("prog1.so")

doStuff <-
    function (nIters, timeInSecs, propBurnIn)
{
    .C("do_stuff",
       as.integer(nIters),
       as.numeric(timeInSecs),
       as.numeric(propBurnIn))
}

doStuff(100, 10, 0.1)
```


The .Call Interface II

- you could just change the `.C` to `.Call` and you would be good to go (let this live in a file called `prog2.R`):

```
dyn.load("prog2.so")

doStuff <-
    function (nIters, timeInSecs, propBurnIn)
{
    .Call("do_stuff",
          as.integer(nIters),
          as.double(timeInSecs),
          as.double(propBurnIn))
}

doStuff(100, 10, 0.1)
```

- where is the difference? well the `C` side of things (which lies peacefully in a file called `prog2.c`) is a little ...

The .Call Interface III

- lets look at the C side

```
#include <R.h>
#include <Rinternals.h>

/*
 * Lots of stuff for bulding the interface for the
 * typedef MonteCarloSpecs
 */

SEXP
do_stuff (SEXP n_iters, SEXP time_in_secs, SEXP prop_burn_in)
{
        MonteCarloSpecs *mcs = NULL;

        mcs = mcs_new(INTEGER(n_iters)[0],
                      REAL(time_in_secs)[0],
                      REAL(prop_burn_in)[0]);
        mcs_print(mcs);
        mcs_free(&mcs);
        return R_NilValue;
}
```


The .Call Interface IV

- points to note about the `do_stuff` function:
  - `C` functions called by `R` must all return `SEXP` (a `typedef` to be introduced later)
  - all arguments passed to the `C` function are passed as `SEXP` type variables and so be *extra* careful not to derefence the arguments directly because we don't know how `SEXP` looks like, use macros (preview coming later), to be found in `Rinternals.h`, instead
  - note in addition to `#include <R.h>` we need `#include <Rinternals.h>`
  - if you don't want to return anything return `R_NilValue`, a special `SEXP` type variable

The `.Call` Interface V

- the only limitation of the `.Call` interface is that it can only have atmost 65 arguments
- usually you would deal with much smaller number of arguments but just in case you happen to use up all 65 of them consider the `.External` interface instead

The .Call Interface VI

- a snapshot of the SEXP typedef, to be found e.g. in `R-2.1.0/include/Rinternals.h`

```
/* The standard node structure consists of a header followed by the
   node data. */
typedef struct SEXPREC {
    SEXPREC_HEADER;
    union {
        struct primsxp_struct primsxp;
        struct symsxp_struct symsxp;
        struct listsxp_struct listsxp;
        struct envsxp_struct envsxp;
        struct closxp_struct closxp;
        struct promsxp_struct promsxp;
    } u;
} SEXPREC, *SEXP;
```

- so SEXP is nothing but a pointer to `struct SEXPREC` which is a complicated animal
- now lets look at `R-2.1.0/include/Rinternals.h`

The .Call Interface VII

- note the "header" SEXPREC_HEADER is nothing but a macro and it tells us that struct SEXPREC is a "doubly-linked" list

```
/* Every node must start with a set of sxpinfo flags and an attribute
   field. Under the generational collector these are followed by the
   fields used to maintain the collector's linked list structures. */
#define SEXPREC_HEADER \
    struct sxpinfo_struct sxpinfo; \
    struct SEXPREC *attrib; \
    struct SEXPREC *gengc_next_node, *gengc_prev_node
```

- the keyword union above is a C construct and I didn't talk about it in C lectures because you would rarely find yourself using this, for the time being lets ignore it, we wouldn't need it for our purposes

The .Call Interface VIII

- some useful macros we will be using all the time:

```
#define CHAR(x)          ((char *) DATAPTR(x))
#define INTEGER(x)       ((int *) DATAPTR(x))
#define REAL(x)          ((double *) DATAPTR(x))
#define STRING_ELT(x,i) ((SEXP *) DATAPTR(x))[i]
#define VECTOR_ELT(x,i) ((SEXP *) DATAPTR(x))[i]

#define CAR(e)           ((e)->u.listsxp.carval)
#define CDR(e)           ((e)->u.listsxp.cdrval)

#define PROTECT(s)       protect(s)
#define UNPROTECT(n)     unprotect(n)
```

Code Files

```
prog2.c
prog2.R
```

