I notice that sometimes the odds in Statistics does not support pending breakpoint. Here is a more general way of activating GDB:

You can using GDB to debug C following the procedures below,

1) Compile the C code (e.g. foo.c) using

```
MAKEFLAGS="CFLAGS=-g" R CMD SHLIB foo.c
```

2) Start R using

```
R -d gdb
```

3) Type ‘R’ to run R.
4) Press ‘Ctrl+C’ to interrupt R.
5) Set the breakpoint at your function (e.g. foo).

```
(gdb) b foo
```

6) Type ‘signal 0’ to return to R and work as usual.

**Structure**

Arrays are used to store large set of data and manipulate them but the disadvantage is that all the elements stored in an array are to be of the same data type. If we need to use a collection of different data type items it is not possible using an array. When we require using a collection of different data items of different data types we can use a structure. Structure is a method of packing data of different types. A structure is a convenient method of handling a group of related data items of different data types.

**Define a structure**

A structure type is defined by

```
struct struct-name {
    type field-name;
    type field-name;
    ...
}
```

e.g.

```
struct student{
    int course;
    int ID;
    double HW1;
    double HW2;
}
```

the keyword struct declares a structure to holds the details of four fields namely course, ID, HW1 and HW2. These are members of the structures. Each member
may belong to different or same data type. The tag name can be used to define objects that have the tag names structure. The structure we just declared is not a variable by itself but a template for the structure. We can declare structure variables using the tag name anywhere in the program. For example the statement,

```c
struct student std1, std[2];
```

declares std1 as variable of type struct student each declaration has four elements of the structure student. Also we can define an array of structure. A structure is usually defines before function. In such cases the structure assumes global status and all the functions can access the structure. Because it is tedious to have to remember to use the word "struct" in these, the structure is often "typedef"-ed to avoid this:

```c
typedef struct student_type {
    char name[20];
    int ID;
} student_type;

student_type std_1;
strcpy(std_1.name, "Bill");
std_1.ID=321;
```

Access the elements of a structure
You access fields of a structure with the \".\" notation, e.g.

```c
std1.course=598;
```

It is common to have pointers to structures. The straightforward notation is clumsy, so a shorthand is available

```c
(*std1_ptr).ID = 123
/*Or */
std1_ptr->ID = 123
```

Like other data type we can initialize structure when we declare them. As for initialization goes structure obeys the same set of rules as arrays we initialize the fields of a structure by the following structure declaration with a list containing values for each fields as with arrays these values must be evaluate at compile time. Here is an example:

```c
struct student std2 = {598, 789, 8.0, 10.0};
```
Here is the code:

```c
#include <stdio.h>
#include <string.h>

struct student{
    int course;
    int ID;
    double HW1;
    double HW2;
};

main()
{
    int i;
    struct student std1, *std1_ptr;
    std1.course=598;

    std1_ptr=&std1;
    (*std1_ptr).HW1=9.5;
    std1_ptr->HW2=10;
    /* And you can initialize the value in this way. */
    struct student std2 ={
        598,
        789,
        8.0,
        10.0
    };

    /* Array of structure */
    struct student std[2];

    for(i=0; i<2; i++) {
        std[i].course=598;
    }

    typedef struct student_type {
        char name[20];
        int ID;
    } student_type;
    student_type std_1;
    strcpy(std_1.name, "Bill");
    std_1.ID=321;
}
```
Linked list

Linked lists and arrays are similar since they both store collections of data. So one way to think about linked lists is to look at how arrays work and think about alternate approaches.

Arrays are probably the most common data structure used to store collections of elements. In most languages, arrays are convenient to declare and the provide the handy [ ] syntax to access any element by its index number. And the entire array is allocated as one block of memory. Each element in the array gets its own space in the array. Any element can be accessed directly using the [ ] syntax. And the address of an element is computed as an offset from the start of the array which only requires one multiplication and one addition. So the disadvantages of arrays are:

1) The size of the array is fixed.
2) Because of (1), the most convenient thing for programmers to do is to allocate arrays which seem "large enough" (e.g. 100 elements). Although convenient, this strategy has two disadvantages: (a) most of the time there are just 20 or 30 elements in the array and 70% of the space in the array really is wasted. (b) If the program ever needs to process more than 100 scores, the code breaks.
3) Inserting new elements at the front is potentially expensive because existing elements need to be shifted over to make room.

Linked lists have their own strengths and weaknesses, but they happen to be strong where arrays are weak. The array's features all follow from its strategy of allocating the memory for all its elements in one block of memory. Linked lists use an entirely different strategy. But linked lists allocate memory for each element separately and only when necessary. An example of linked list structure would be

```c
struct node {
    int data;
    struct node* next;
};
```

Each node contains a single data set element and a pointer who points to the next node in the list.