Introduction to GDB

The GNU Debugger, usually called just GDB and named gdb as an executable file, is the standard debugger for the GNU software system. It is a portable debugger that works for many programming languages, including Ada, C, C++, Objective-C, and Pascal. It allows you to see what is going on `inside' another program while it executes -- or what another program was doing at the moment it crashed.

GNU Debugger helps you in finding out followings:

1) If a core dump happened then what statement or expression did the program crash on?
2) If an error occurs while executing a function, what line of the program contains the call to that function, and what are the parameters?
3) What are the values of program variables at a particular point during execution of the program?
4) What is the result of a particular expression in a program?

How GDB Debugs?

GDB allows you to do things like run the program up to a certain point then stop and print out the values of certain variables at that point, or step through the program one line at a time and print out the values of each variable after executing each line.

GDB uses a simple command line interface.

How it works?

To let GDB be able to read all that information line by line from the symbol table, we need to compile it a bit differently. Normally we compile things as:

```
gcc hello.c
```

Instead of doing this, we need to compile with the -g flag as such:

```
gcc -g hello.c
```

It will Compile hello.c with the debugging option (-g). You still get an a.out, but it contains debugging information that lets you use variables and function names inside GDB, rather than raw memory locations. Then use the following command to Open GDB with file a.out, but does not run the program.

```
gdb a.out
```

There are three ways to run “a.out”, loaded previously. You can run it directly

```
(gdb) r
```

pass arguments

```
(gdb) r arg1 arg2
```

or feed in a file

```
(gdb) r < file1
```

You will usually set breakpoints before running.

There is a big list of GDB commands but following commands are among the more useful gdb commands:

- b main - Put a breakpoint at the beginning of the program
- b - Put a breakpoint at the current line
- b N - Put a breakpoint at line N
- b +N - Put a breakpoint N lines down from the current line
- b fn - Put a breakpoint at the beginning of function "fn"
- d N - delete breakpoint number N
- info break - list breakpoints
- r - Run the program until a breakpoint or error
- c - continue running the program until the next breakpoint or error
- f - Run until the current function is finished
- s - run the next line of the program
- s N - run the next N lines of the program
- n - like s, but don't step into functions
- p var - print the current value of the variable "var"
- q - Quit gdb

Try to use GDB to debug the following codes

```
#include <stdio.h>

/* This code contains some logical errors. Try using GDB to find out */
/* For two given 2 by 1 vectors 1 and 2, compute the inner product */
double inner(double *a, double *b, int n);

main()
{
        int i, j;
        double a[2][2], r;
        for(i=0; i<2; i++) {
                printf("Vector %d:\n", (i+1));
                printf("1st element:");
                scanf("%f", &a[i][1]);
                printf("2nd element:");
                scanf("%f", &a[i][2]);
        }
        r = inner(a[0], a[1], 2);
        printf("The inner product is %f\n", r);

}
```

```
double inner(double *a, double *b, int n) {
	int i, res;
	for(i=0; i<n; i++) {
		res = res + a[i]*b[i];
	}
	return res;
}
```

Another example,

```
#include <stdio.h>
#include <stdlib.h>

/* This code contains some errors, use GDB to find them. */

void count(FILE *fp, int *size);

main()
{
	int *size;
	FILE *fp=fopen("test.txt","r");
	if (fp == NULL) {
		fprintf(stderr, "file does not exist\n");
		return 0;
	}

	count(fp, size);
	printf("The # of element in the file is %d\n", *(size));
}

void count(FILE *fp, int *size)
{
	*size=0;
	double c;
	while( fscanf(fp, "%lf", &c)!=EOF ) {
		*(size)++;
	}
}
```

Calling C functions from R

R is rotten at iterative algorithms that require loops iterated many times (especially Markov chain Monte Carlo), not as bad as S-PLUS, but still bad. A way to get all the speed advantages of C is to write the inner loop in C and call it from R. Here is a very simple C function

```
void foo(int *nin, double *x)
{
        int n = nin[0], i;
        for (i=0; i<n; i++)
                x[i] = x[i] * x[i];
}
```

Notice two properties that are required for a function callable from R

- The function does not return a value. All work is accomplished as a "side effect" (changing the values of arguments).
- All the arguments are pointers. Even scalars are vectors (of length one) in R.

Compiling and Dynamic Loading

Under Linux:

Put the C example code in a file foo.c and compile it to a shared library using the command

```
R CMD SHLIB foo.c
```

Now the code can be dynamically loaded into R by doing (in R)

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

The Call to C

The actual call to C from R is made by the R function .C like this

```
.C("foo", n=as.integer(3), x=as.double(c(1,2,3))
```

Under Windows:

First you are going to have to install some tools that will allow you to build R packages and compile C code from the command line. Luckily all of the tools you will need have been bundled together for you as Rtools.exe.

Once you have downloaded and installed those tools, you will need to change the PATH of your environment variables. I think this is necessary so that the R tools you just installed can be called from within R. We need to redefine the path such it that includes (I pulled the following from the readme file that is associated with the Rtools.exe file: Rtools.txt):

PATH=c:\Rtools\bin;c:\Rtools\perl\bin;c:\Rtools\MinGW\bin;c:\R\bin;
c:\Rtools\MinGW64\bin;(if you're using 64 bit system)<others>.For all applications via Windows, how you set an environment variable is system specific: under Windows 2000/XP/2003 you can use `System' in the control panel or the properties of `My Computer' (under the `Advanced' tab). Under Vista, go to `User Accounts' in the Control Panel, and select your account and then `Change my environment variables'."

Once you are in the change environment variable box, select "New". Name the new variable "PATH" and then set the value to (at a minimum):

c:\Rtools\bin;c:\Rtools\perl\bin;c:\R\R-2.11.0\bin c:\Rtools\MinGW\bin;
c:\Rtools\MinGW64\bin;(if you're using 64 bit system)
Then, you're ready to go.

Debugging C called from R using GDB

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

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

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

2) Start R using

```
R -d gdb
```

3) Set the breakpoint at your function (e.g. foo), even it's still pending

```
(gdb)  b foo
```

4) Type 'R' to run R and work as usual.