Debugging Techniques

CEFET Engineering Week

Petrópolis, May 10th 2017

Luís Tarrataca

Task 1

It is expected that course attendees will have knowledge about the C language and data structure development. For the purpose of this minicourse we will focus on developing C code for a doubly linked list. Doubly Linked List is a variation of Linked list in which navigation is possible in both ways, either forward and backward easily as compared to Single Linked List.

Doubly linked list offers easy implementation of many operations, whereas singly linked list requires more info for the same operation. For example, the deletion of a node in a singly linked list. we need the pointer to node previous to it. But in case of doubly linked list, we just need the pointer to the node being deleted. A node on a singly linked list cannot be removed unless we have the pointer to its predecessor. However, doubly linked lists occupy more space and often require more operations for similar tasks when compared to singly linked lists.

This data structure requires a level of detail that is sure to introduce some bugs during development. Please proceed by implementing a doubly linked list in accordance with the structure definitions and function prototypes stated below:

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


/* STRUCTURE DEFINITIONS */
typedef struct _Element{

     int value;

     struct _Element* prev;
     struct _Element* next;

}Element;

typedef struct _DoubleLinkedList{

     Element* first;
     Element* last;

}DoubleLinkedList;


/* FUNCTION PROTOTYPES */
DoubleLinkedList* new();

Element* insert( int value, DoubleLinkedList* list );

Element* delete( int value, DoubleLinkedList* list );

void destroy( DoubleLinkedList* list );

void print( DoubleLinkedList* list );
```

Task 1 continued on next page. . .

Start by implementing the creation and insertion functions. During the development make sure to compile the code using the "-g" option, which allows for use of symbols required for the debugging procedure:

```
$ gcc -c -g DoubleLinkedList.c
$ gcc -o DoubleLinkedList DoubleLinkedList.o
$ ./DoubleLinkedList
```

Once you feel satisfied with your code you can start testing by perform some insertions of random numbers, using the code below. Although doubly linked lists are an elementary data structure it is not uncommon for the initial versions of the code to contain some bugs. We will find these out using the debugging procedure described later on.

```
int main( int argc, char** argv){

     // Random number generator initialization
     srand( time( NULL ) );

     int randomNumber = 0;

     DoubleLinkedList* list = new();

     if( list == NULL ){ return EXIT_FAILURE; }

     for( int counter = 0; counter < 10; counter++ ){

          // returns a pseudo-random integer between 0 and RAND_MAX
          randomNumber = rand();

          insert( randomNumber, list );

      }

      print( list );
}
```

Task 2

Does your initial code contain bugs? Probably yes. So lets try to use the “gdb” debugger that can be installed in Linux. This is a text-based debugger which allows you to see the state of your program at any given time. You can launch the C debugger (GDB) via a terminal as shown below:

```
$ gdb DoublyLinkedList
```

GDB works using a concept called *break points*, which are code lines of interest in our code where we suspect errors. By adding break points to the code we can analyze the state of our program immediately before the execution of the line where the break point was added. While executing the program, the debugger will stop at the break point, and gives you the prompt to debug. Break points can be added at any time during the debugging session. However, it is common practice to add some initial break points before we start the execution of the program. This can be done through the two following gdb instructions:

- break line_number
- break function_name

In our example lets start by adding a break point to the main function:

```
(gdb) break main
```

You can start running the program using the run command in the GDB debugger:

```
(gdb) run
```

Once you execute the C program, it will execute until the first break point, and give you the prompt for debugging. You can use the following gdb commands to debug:

- list: List source code.
- print variable_name: To see the state of a variable;
- continue: Debugger will continue executing until the next break point.
- next: Debugger will execute the next line as single instruction.
- step: Same as next, but does not treats function as a single instruction, instead goes into the function and executes it line by line.

It is also important to mention that pressing the enter key will execute the previously executed command again. We are now in a position to start executing multiple lines of code (by repeatedly pressing “n”) and checking the state of the variables, *e.g.*:

```
(gdb) print randomNumber
(gdb) print list
(gdb) print list->first
```

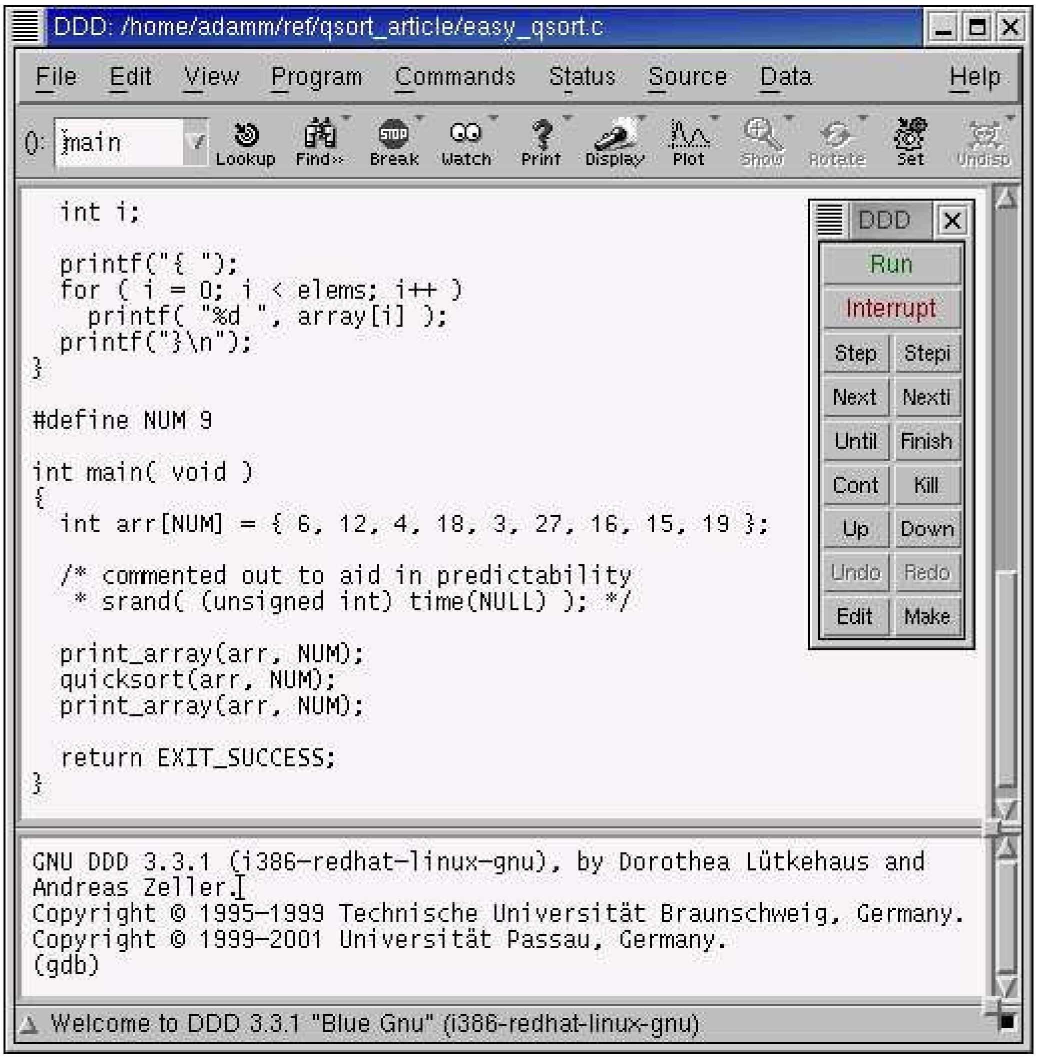
Task 2 continued on next page. . .

Task 3

GDB seems like a nice way to decode your code, right? However, debugging would be made easier if we had some type of user interface. Fortunately, this can be done using the “ddd” linux program. The Data Display Debugger (ddd) is a graphical user interface for command-line debuggers such as GDB.

DDD has GUI front-end features such as viewing source texts and its interactive graphical data display, where data structures are displayed as graphs. A simple mouse click dereferences pointers or views structure contents, updated each time the program stops. Using DDD, you can reason about your application by watching its data, not just by viewing it execute lines of source code. DDD accepts exactly the same commands that you would provide in GDB (run, print, list, etc...) making it easier to analyze your code. It can be done by executing the “ddd” command in a terminal window:

```
$ ddd DoublyLinkedList
```


DDD: /home/adamm/ref/qsort_article/easy_qsort.c
File Edit View Program Commands Status Source Data Help
(): main
Lookup Find» Break Watch Print Display Plot Show Rotate Set Undisp
DDD
Run Interrupt Step Stepi Next Nexti Until Finish Cont Kill Up Down Undo Redo Edit Make

```
  int i;

  printf("{ ");
  for ( i = 0; i < elems; i++ )
    printf( "%d ", array[i] );
  printf("}\n");
}

#define NUM 9

int main( void )
{
  int arr[NUM] = { 6, 12, 4, 18, 3, 27, 16, 15, 19 };

  /* commented out to aid in predictability
   * srand( (unsigned int) time(NULL) ); */

  print_array(arr, NUM);
  quicksort(arr, NUM);
  print_array(arr, NUM);

  return EXIT_SUCCESS;
}
```

```
GNU DDD 3.3.1 (i386-redhat-linux-gnu), by Dorothea Lütkehaus and
Andreas Zeller.
Copyright © 1995-1999 Technische Universität Braunschweig, Germany.
Copyright © 1999-2001 Universität Passau, Germany.
(gdb)
```

Welcome to DDD 3.3.1 "Blue Gnu" (i386-redhat-linux-gnu)

Figure 1: The DDD user interface

Task 3 continued on next page...

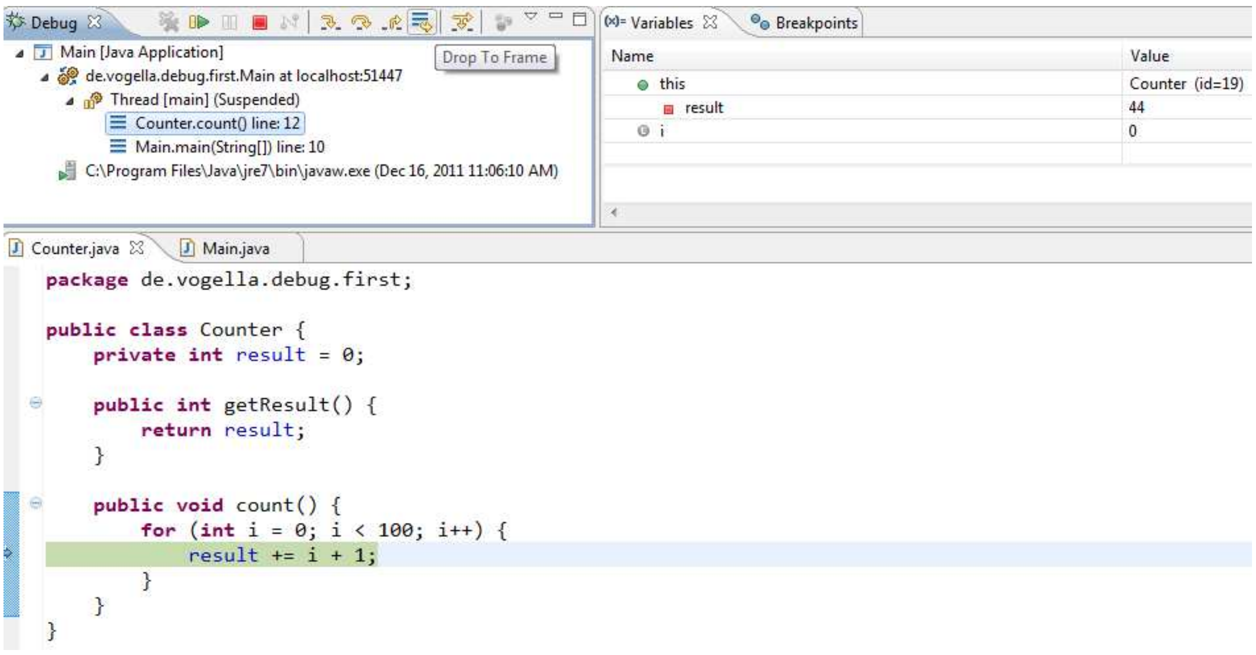
You can then proceed by visually adding breakpoints in the code window, or by simply executing the previous GDB commands:

```
(gdb) break main
(gdb) run
(gdb) print randomNumber
(gdb) print list
(gdb) print list->first
```

Task 4

Ok, so now you know how to debug code through GDB and DDD. These are two convenient mechanisms for debugging. However, the process can still be improved an integrated development environment (IDE). An IDE is a software application that provides comprehensive facilities to computer programmers for software development.

An IDE normally consists of a source code editor, build automation tools and a **debugger**. Most modern IDEs have intelligent code completion. Some examples of IDEs include Eclipese, NetBeans, IntelliJ IDEA, SharpDevelop and Lazarus. The following screenshot represents the debugger interface from Eclipse:



```
Main [Java Application]
  de.vogella.debug.first.Main at localhost:51447
    Thread [main] (Suspended)
      Counter.count() line: 12
      Main.main(String[]) line: 10
  C:\Program Files\Java\jre7\bin\javaw.exe (Dec 16, 2011 11:06:10 AM)
```

Name	Value
this	Counter (id=19)
result	44
i	0

```
package de.vogella.debug.first;

public class Counter {
    private int result = 0;

    public int getResult() {
        return result;
    }

    public void count() {
        for (int i = 0; i < 100; i++) {
            result += i + 1;
        }
    }
}
```

Figure 2: The Eclipse debugger user interface

Unfortunately, there will be no time in this mini course to discuss the Eclipse IDE, which by itself would require a separate course. However, it is important to say that most developers use some type of IDE in their professional careers given the vast set of tasks they facilitate. Therefore, in your spare time, you can choose one IDE and start using it in your university projects. Consider this your homework ;)