CSCI 2270 Data Structures and Algorithms
Instructor: Hoenigman / Jacobson
Recitation 13

In this recitation, we will learn about the basics of inserting into a graph and how to debug your program using gdb.

Graphs[1]

Graphs provide a structure for representing connections between people, places, or things that captures the essence of the connections. In the graph ADT, the vertices in the graph are stored as a private variable. There are public methods to initialize the graph, insert and delete edges and vertices, print the graph, and search the graph. The edges are stored in an adjacency list for each vertex in the vertices variable. We will only be looking at Inserting for a quick review on graphs.

```
1.    private:
2.        vertices
3.    public:
4.        Init()
5.        insertVertex(value)
6.        insertEdge(startValue, endValue, weight)
```

Creating graph vertices and edges

This graph implementation uses vectors instead of an array or a linked list to simplify the memory management of the graph.

To declare a vector variable:
vector<type> variable;

In code, the graph can be represented in a Graph class:

```
class Graph{
  private:
        //vertices and edges definition goes here
  public:
        //methods for accessing the graph go here
}
```

Each vertex in the graph is defined by a struct with two members: a key that serves as the key value for the vertex and a vector, adjacent, to store the adjacency list for the vertex.

```
struct vertex{
        string key;
        vector<adjVertex> adjacent;
}
```

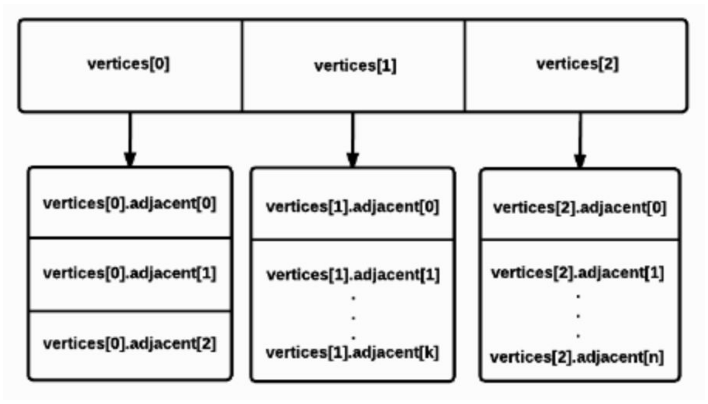
[1] Excerpt From: Rhonda Hoenigman. “Visualizing Data Structures.” iBooks.

The adjVertex data type is also defined by a struct with two members: contains a pointer to the adjacent vertex v, and an integer weight that stores the edge weight between the two vertices.

```
struct adjVertex{
        vertex *v;
        int weight;
};
```

An empty vector of vertex can be created using the statement:
std::vector <vertex> vertices;

The adjVertex struct only stores the destination vertex in v because the origin vertex is stored in the vertices vector. In this design, each vertex in the graph has a vector of adjacent vertices that contains the vertex at the other end of the edge. The number of adjacent vertices can vary for each vertex in vertices. The size of the adjacent vector is also dynamic for each vertex.



A visual representation of the vertices vector and the adjacent vector for each vertex. The vertices vector contains all of the vertices in the graph. The adjacent vector is the adjacency list for each vertex.

Insert vertex

Adding vertices to the graph is handled through a public method that takes the vertex key value as an argument. The insertVertex() adds a vertex with the specified value to a graph.

Pre-conditions

value is a valid key value of the same type as the key parameter in vertex.
vertices is an array of graph vertices.

Post-conditions

Vertex added to vertices if it doesn't already exist.

Algorithm

```
void Graph::insertVertex(string value){
1.  bool found = false;
2.  for(int i = 0; i < vertices.size(); i++){
3.      if(vertices[i].key == value){
```

```
            found = true;
            cout<<vertices[i].key<<" found."<<endl;
           break;
        }
  }
  if(found == false){
        vertex v;
        v.key = value;
        vertices.push_back(v);
  }
}
```

The insertVertex() method can be called within main() as follows:

```
Graph g;
g.insertVertex("Boulder");      //create an instance of graph and add a vertex with the key
Boulder
```

Insert edge

After vertices have been added to the graph, edges can be added to connect them. The insertEdge(), takes the two key values of the vertices to connect and the weight of the edge between them, and adds an element to the adjacent list for the source vertex.

Pre-conditions

v1 and v2 exist in the graph and there isn't an existing edge from v1 to v2.

Post-conditions

Entry added to the adjacency list for v1 connecting it to v2 with the specified weight.

```
void Graph::insertEdge(string v1, string v2, int weight){
  for(int x = 0; x < vertices.size(); x++){
      if(vertices[x].key == v1){                        // looking for v1
          for(int y = 0; y < vertices.size(); y++){
              if(vertices[y].key == v2 && x != y){             // looking for v2
                  adjVertex av;
                  av.v = &vertices[y];                  // av points to the destination (v2) address
                  av.weight = weight;                   // updating the av weight
                  vertices[x].adjacent.push_back(av);
              }
          }
      }
  }
}
```

Note: The insertEdge() method adds an edge in one direction only. In an undirected graph, the method would need to be called twice with the source and destination vertices swapped to add the edge between two vertices in both directions. For example, to add an undirected edge between Boulder and Denver:

```
g.insertEdge("Boulder", "Denver", 30);
g.insertEdge("Denver", "Boulder", 30);
```

calls insertEdge() the first time with Boulder as the source and Denver as the destination and calls insertEdge() the second time with Denver as the source and Boulder as the destination.

Debugging your C++ program

Debugging is an important part of programming and the development cycle. The process of *debugging* is to identify the source of the problem and a way to correct or work around it to run your program properly. This process should start as soon as the code is written and should continue throughout. As programs grow, it becomes more difficult to debug if not initiated early. With a working knowledge of a debugger, like gdb, finding and fixing errors in your program will go much faster.

A debugger is a program that runs other programs, allowing the user to exercise control over these programs and to examine variables when problems arise. Though there are multiple strategies used to debug a program, we will be looking at gdb. GNU Debugger, known as **gdb,** is the most popular debugger for UNIX systems to debug C and C++ programs. You will need to make sure you are using the correct system in order to use this programming tool.

GNU Debugger helps you in getting information about the following:

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

Installing gdb

You can check if gdb is installed by typing the following in your terminal:

```
$ gdb -help
```

If it's not installed, then you can type the following in your terminal (using your linux machine).

```
$ sudo apt-get install libc6-dbg gdb valgrind
```

To double check it was installed, repeat the help command. Once installed, it will provide you with an available menu option that can be used with gdb.

Useful commands

Throughout the write-up you will notice we cover several commands used within gdb. In case you prefer an advance list, here is a helpful website:
https://www.cs.rochester.edu/~nelson/courses/csc_173/review/gdb.html

Debugging a program the produces a core dump

In this program, we are trying to divide a number by 0, which is bound to result in a crash. Using this example, let's see how we can debug the code. *Note:* In a more complicated program it might be much harder to find the source of a crash/core dump, such as a seg fault.

Example:

```
int divint(int, int);
int main() {
  int x = 5, y = 2;
  cout << divint(x, y);
  x =3;

  y = 0;
  cout << divint(x, y);
  return 0;
}

int divint(int a, int b) {
  return a / b;
}
```

When we compile this program and run it normally, it will produce: Floating point exception: 8. This is a runtime error in which you can't divide by zero. So, let's say this program is named as gdbtest.cpp. We will compile and run the program:

```
$g++ gdbtest.cpp -o gdbtest
$./gdbtest
```

Produces:

```
Floating point exception (core dumped)
```

In order to degub using gdb, it's recommended to use the '-g' flag when compiling the program as follows. Note: By compiling the code with the special flag, it will allow the compiler to collect the debugging information. Basically, it will provide you with more information.

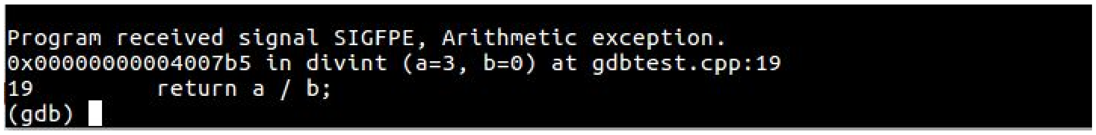
```
$ g++  -g -std=c++11 gdbtest.cpp -o gdbtest
```

In order to start running a program within gdb, we use the following command. This will open a gdb terminal:

```
$ gdb  ./gdbtest
```

In order for the GNU debugger to run the code, we use the command "run" or "r". Your program will run until termination, a breakpoint is reached, or an error occurs. In the case below, we have not set any breakpoints and will run until the program crashes/errors. r*un* will also restart the program when stopped in mid-execution.

(gdb) run



```
Program received signal SIGFPE, Arithmetic exception.
0x00000000004007b5 in divint (a=3, b=0) at gdbtest.cpp:19
19          return a / b;
(gdb) 
```

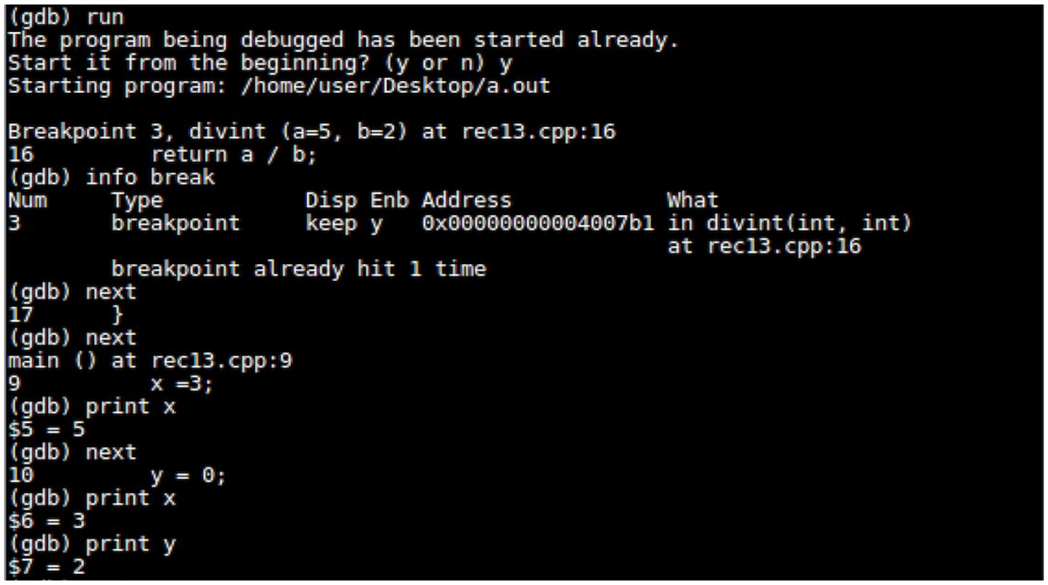
Line 19 represents the line number at which the code crashes and will also print the line of code corresponding to it. Here, a and b values were set to 3 and 0 respectively at the time of the crash. If we had set a breakpoint before running, then it would stop when divint() is called.

(gdb) break
break sets breakpoints at places where you want the debugger to stop. You can set multiple breakpoints.
break function-name will set a breakpoint at the start of the function.
break program-name.cpp:function-name will set a breakpoint on the first line within the function
break line-number will set a breakpoint at a particular line number within the program.

(gdb) next
You will also notice in the below example, *next*. It will run until the next source line is reached and used to move to the next line without needing to step through a function. When using next, then line isn't executed until after it has passed.

(gdb) print
Will display the last variable or expression.

For example:



```
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/user/Desktop/a.out

Breakpoint 3, divint (a=5, b=2) at rec13.cpp:16
16          return a / b;
(gdb) info break
Num     Type           Disp Enb Address            What
3       breakpoint     keep y   0x00000000004007b1 in divint(int, int)
                                                   at rec13.cpp:16
        breakpoint already hit 1 time
(gdb) next
17      }
(gdb) next
main () at rec13.cpp:9
9           x =3;
(gdb) print x
$5 = 5
(gdb) next
10          y = 0;
(gdb) print x
$6 = 3
(gdb) print y
$7 = 2
```

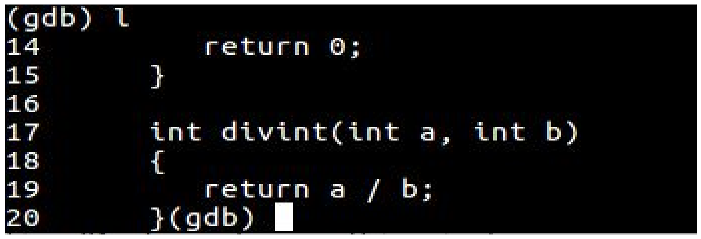
In this example, the breakpoint was set previously as noted above. When running the program, it stopped on line 16, which is the first line of the function.

With **(gdb) info break** we are allowed to see the information about the breakpoints. Here, the breakpoint 3 is located at that memory address listed and the function you are breaking on at the particular line number within the file. Then, we continue using *next,* to see the next line, and *print*, to see the value of each variable.

(gdb) l

l is short for 'list' which is useful for seeing the context of the crash. It lists the surrounding lines of code lines near the 19th line, which is the main source of the crash. Since this is a small program, only a few lines are listed, but usually there are approx 10 lines listed.

For example:

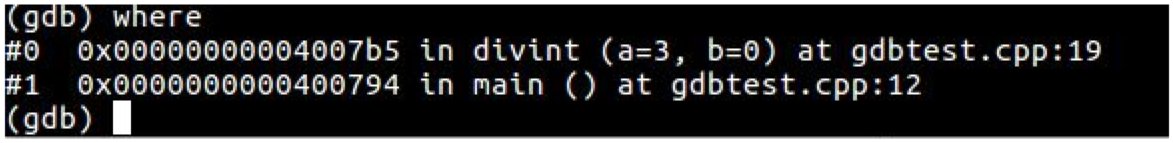


```
(gdb) l
14          return 0;
15      }
16
17      int divint(int a, int b)
18      {
19          return a / b;
20      }(gdb)
```

(gdb) where

where is equivalent to **'bt' or backtrace** and allows you to find out where your program is or where it failed/seg faulted. Produces what is known as a 'stack trace'. Since, main() is called first, then it will be at the bottom of the stack, then divint() is called and pushed onto the stack. You should read this as follows: The crash occurred in the function divint at line 19. This, in turn, was called from the function main() at line 12.

For example:



```
(gdb) where
#0  0x00000000004007b5 in divint (a=3, b=0) at gdbtest.cpp:19
#1  0x0000000000400794 in main () at gdbtest.cpp:12
(gdb)
```

(gdb) delete

Will delete your breakpoints

In Class work: Debugging logical errors

For the in class work, you will download the recitationInClass13.cpp[2] on moodle and will walk through a trace as we have just completed. This program has logical errors that we need to investigate. We will get you started and you will try to finish the debugging process and trace the errors. Currently, the program outputs a value of infinity, regardless of the inputs. When completed, the program is supposed to output the value of the following series:

(x^0)/0! + (x^1)/1! + (x^2)/2! + (x^3)/3! + (x^4)/4! + ... + (x^n)/n!   // given x and n as inputs.

[2] http://cs.baylor.edu/~donahoo/tools/gdb/tutorial.html
https://www.tutorialspoint.com/gnu_debugger/gdb_quick_guide.htm

When you compile and run the program, the following is what you will get:

$ g++ logicalerror.cpp -o logicalerror
$ ./logicalerror

Produces:

```
The value of the series is inf
```

The value for any value of x and n is always inf. Now, let's start debugging the program to understand why we get a value of infinity each time.

g++ -g -std=c++11 logicalerror.cpp -o logicalerror
gdb ./logicalerror

(gdb) break <line number>

```
(gdb) break 30
Breakpoint 1 at 0x40094e: file logicalerror.cpp, line 30.
```

To set a breakpoint at line 30 is when computeseries() is being called from the main function. The program will pause when it reaches the breakpoint. With reasoning, we know that this is the first call from main() and need to stop it right after it runs.

(gdb) run

```
(gdb) run
Breakpoint 1, main () at logicalerror.cpp:30
warning: Source file is more recent than executable.
30      double seriesValue = ComputeSeriesValue(2, 3);
```

After setting the breakpoint, run the debugger. It will run until the breakpoint.

Now that you've executed the the program till line 30, step into the function computeSeriesValue() to understand what's going wrong.

(gdb) step

```
(gdb) step
ComputeSeriesValue (x=2, n=3) at logicalerror.cpp:17
17      double seriesValue = 0.0;
```

Runs to the next instruction, not to the next line (as it does when using next). If the current instruction is setting a variable, it is the same as next. If it's a function, it will jump into the function, execute the first statement, and then pause. step is good for diving into the details of your code.

At this point, the program control is at the first statement of the function ComputeSeriesValue (x=2, n=3). From here, you will hit next to see each line within the function.

(gdb) next

```
(gdb) next
18      double xpow = 1;
```

This runs the program until next line, then pauses. If the current line is a function, it executes the entire function, then pauses. **next** is good for walking through your code quickly.

Since, we are inside the function computeSeriesValue, the command next would execute the next line after double SeriesValue=0.0

<u>(gdb) bt</u>
If you want to know where you are in the program's execution (and how, to some extent, you got there), you can view the contents of the stack using the backtrace command as follows:

```
(gdb) bt
#0  ComputeSeriesValue (x=2, n=3) at logicalerror.cpp:18
#1  0x000000000040096b in main () at logicalerror.cpp:30
```

<u>Continuing on using gdb:</u>
Now, let’s keep going to the next lines and step into the computeFactorial function:

```
(gdb) next
20      for (int k = 0; k <= n; k++) {
(gdb) next
21        seriesValue += xpow / ComputeFactorial(k);
(gdb) step
ComputeFactorial (number=0) at logicalerror.cpp:7
7       int fact = 0;
(gdb) next
9       for (int j = 1; j <= number; j++) {
(gdb) next
13      return fact;
(gdb) print fact
$1 = 0
(gdb) quit
A debugging session is active.
      Inferior 1 [process 8651] will be killed.
Quit anyway? (y or n) y
```

The **print command** (abbreviated p) reveals that the value of *fact* never changes.
<u>Note</u> that the function is returning a value of 0 for the function call ComputeFactorial(number=0). This is an error. By taking a closer look at the values printed above, we realize that we are computing fact = fact * j where fact has been initialized to 0; fact should have been initialized to 1. We can now quit GDB with the quit command and change the following line: int fact = 1;

<u>Recitation Exercise</u>
For the programming exercise, you will need to download the Recitation13.cpp. This program is on graphs and has known errors, which you will fix them using gdb. You will zip together screenshots using gdb when the error is found and your Recitation13.cpp file with a brief explanation on how you fixed it. You will have until Sunday at 5pm to complete the exercise.