C++ Debugging Guide

University of Pretoria

COS 110

Claudio Da Silva

GDB, Valgrind and other debugging methods
Debugging without tools

Sometimes an error is too small to sit through an entire debugging process, or you end up in a situation where you do not have the required tools available in order to debug your program. In this case debugging should come down to one simple concept, a print statement such as cout or printf.

What happens in a segmentation fault is it causes the program to crash, mainly for the safety of your own system. This means that the fault only occurs after everything that has run correctly before it has already occurred.

This means that anything that is printed before the segmentation fault happened will still be printed to the terminals output, and any code after the segmentation fault line will not execute, including print statements.

By using a cout statement before a piece of code that we think is giving a segmentation fault, as well as a cout statement right after that piece of code, we can see if that piece of code was what was causing this segmentation fault to happen.

For example from this code we get:

```cpp
#include <iostream>

using namespace std;

int main()
{
    /* The size we are going to allocate to the pointer, this means we can access index 0 - 9 only. */
    int size = 10;

    /* A double pointer int is created, and initialized to create an array of int pointers of size 10 */
    int** testPointer = new int* [size];

    /* Here we will assign values 0 - 9 to pointer indexes 0 - 9, note the new int() being used to assign an int object and not an actual integer */
    for (int i = 0; i < 10; i++)
    {
        testPointer[i] = new int();
    }

    /* Here by mistake we go to size 10 included, this means we will try to dereference a memory address that was never set, causing a segmentation error */
    for (int i = 0; i <= 10; i++)
    {
        cout << *testPointer[i] << endl;
    }

    return 0;
}
```

claudio@claudio-VirtualBox:~/Desktop/segTest$ make
g++ -c segTest.cpp -o segTest.o
g++ segTest.o -o segTest
claudio@claudio-VirtualBox:~/Desktop/segTest$ ./segTest
0
1
2
3
4
5
6
7
8
9
Segmentation fault (core dumped)
claudio@claudio-VirtualBox:~/Desktop/segTest$
When using the following code however:

```cpp
#include <iostream>
using namespace std;

int main()
{
    /* The size we are going to allocate to the pointer, this means we can access index 0 - 9 only. */
    int size = 10;

    /* A double pointer int is created, and initialized to create an array of int pointers of size 10 */
    int** testPointer = new int* [size];
    cout << "Testing 1" << endl;

    /* Here we will assign values 0 - 9 to pointer indexes 0 - 9, note the new int() being used to assign an int object and not an actual integer */
    for (int i = 0; i < 10; i++)
    {
        testPointer[i] = new int(i);
    }
    cout << "Testing 2" << endl;

    /* Here by mistake we go to size 10 included, this means we will try to dereference a memory address that was never set, causing a segmentation error */
    for (int i = 0; i <= 10; i++)
    {
        cout << *testPointer[i] << endl;
    }
    cout << "Testing 3" << endl;
    return 0;
}
```

We can see clearly in this case that “Testing 1” and “Testing 2” both printed, but “Testing 3” did not. Thus we can clearly assume that the problem comes from the loop between the second and third print statements.
After correcting it, we can see the expected result we would like to receive:

```cpp
#include <iostream>

using namespace std;

int main()
{
    /* The size we are going to allocate to the pointer, this means we can access index 0 - 9 only. */
    int size = 10;

    /* A double pointer int is created, and initialized to create an array of int pointers of size 10 */
    int** testPointer = new int* [size];

    cout << "Testing 1" << endl;

    /* Here we will assign values 0 - 9 to pointer indexes 0 - 9, note the new int() being used to assign an int object and not an actual integer */
    for (int i = 0; i < 10; i++)
    {
        testPointer[i] = new int(i);
    }

    cout << "Testing 2" << endl;

    /* Here by mistake we go to size 10 included, this means we will try to dereference a memory address that was never set, causing a segmentation error */
    // cout << *testPointer[i] << endl;

    /* Here we use the correct version where we go to less than 10 */
    for (int i = 0; i < 10; i++)
    {
        cout << *testPointer[i] << endl;
    }

    cout << "Testing 3" << endl;

    return 0;
}
```

**WARNING:** ALWAYS REMEMBER TO REMOVE TESTING STATEMENTS BEFORE UPLOADING TO FITCHFORK
GDB – The GNU project debugger

GDB is a handy tool built in to most UNIX and Linux systems that allows you to essentially see what is happening inside another program. It allows us to detect problems in our programs such as segmentation faults by tracing the program one line at a time. It also makes it easy to manage how you test by adding backtracking techniques, which allow you to trace back your program from where the error occurred as well as assign breakpoints to test your program from a certain location only.

Source:
http://www.gnu.org/software/gdb

How to install (Provided your system does not come standard with GDB):

- sudo apt-get install gdb (Ubuntu/Debian based systems).
- sudo yum install gdb (Fedora/RedHat based systems).
- For Windows it is better to use an IDE based debugger, but MinGW should allow you to use GDB.

Compile note:

When compiling using the gcc/g++ compiler, add the -g flag to your make. For example g++ -c -g main.cpp -> main.o. What this does is adds debug information to object files created which makes it possible for GDB to go through and find the errors that may occur in your code.

How to run:

Use the commands gdb program in terminal to start GDB, where program is the name of your executable for example ./main. Once GDB is open, use the run or r command to run your program, as you can see in the following screenshot, GDB can help point out the exact line your fault is in:
Commands:

**Start**: Begins a step by step execution of your program.

**Stop**: Ends the step by step execution.

**Step or s**: Takes one step forward in your program execution.

**Continue or c**: Continues to the next set breakpoint.

To add a breakpoint, use **break <line number>** or **b <line number>**. This allows you to set the step in your program you would like to jump to.

You can also add a breakpoint at a specific function using **break <function name>** or **b <function name>**.

When working with multiple files one can set a breakpoint at a specific file's line using **break <file name>:<line number>** or **b <file name>:<line number>** e.g. **break queue.cpp:32**.

**Backtrace**: Back tracks after an error occurs.

**Quit or q**: Allows you to exit the GDB instance you are running.
Valgrind – The memory test program

Valgrind is a tool used in detecting errors in memory management and threading bugs. It can be used to detect things such as memory leaks (These are sections of memory assigned but never correctly deleted thus becoming irrecoverable by the system until restart.), use of variables that have yet to be initialized (Such as a count variable that has a garbage value as it was never set to 0), invalid pointers and double memory deallocation (Deleting of memory that has already been deleted.).

Source:

http://www.valgrind.org

How to install:

- `sudo apt-get install valgrind` (Ubuntu/Debian based systems).
- `sudo yum install valgrind` (Fedora/RedHat based systems).
- Valgrind is not actually designed for windows, but if I’m not mistaken there should be a tool called Valgrind4Win

Compile note:

When compiling using the gcc/g++ compiler, add the –g flag to your make. For example `g++ -c -g main.cpp -o main.o`. What this does is adds debug information to object files created which makes it possible for Valgrind to go through and find the errors that may occur in your code.

How to run:

Use the commands `valgrind program` in terminal to start Valgrind, where `program` is the name of your executable for example `./main`. Running using the –v flag we see it telling us that there was memory unallocated that we tried to access:

```
==2499==  possibly lost: 0 bytes in 0 blocks
==2499==  still reachable: 120 bytes in 11 blocks
==2499==  suppressed: 0 bytes in 0 blocks
==2499==  Rerun with --leak-check=full to see details of leaked memory
==2499==  ERROR SUMMARY: 2 errors from 2 contexts {suppressed: 0 from 0}
==2499==  1 errors in context 1 of 2:
==2499==  Invalid read of size 4
==2499==  at 0x40095C: main (segTest.cpp:26)
==2499==  Address 0x8 is not stack'd, malloc'd or (recently) free'd
==2499==
==2499==  1 errors in context 2 of 2:
==2499==  Invalid read of size 8
==2499==  at 0x400959: main (segTest.cpp:26)
==2499==  Address 0x5a2f090 is 8 bytes after a block of size 80 alloc'd
==2499==  at 0x4c2c20: operator new[] (unsigned long) (in /usr/lib/valgrind/valgrind-memcheck-amd64-linux.so)
==2499==  by 0x4008FD: main (segTest.cpp:12)
==2499==
==2499==  ERROR SUMMARY: 2 errors from 2 contexts {suppressed: 0 from 0}
Segmentation Fault (core dumped)
```
Flags:

-\textbf{v}: Is for a verbose mode giving a more detailed description of things

--\textbf{leak-check}=<\textbf{parameter}>: If \textbf{no} then no checking will be done, if \textbf{yes} checking will be done, if \textbf{summary} which is the default then it will give a summary of leak checking, if \textbf{full} a full leak report will be given

--\textbf{show-reachable}=<\textbf{Parameter}>: If \textbf{yes} then all blocks will be shown including ones that could be reached, if \textbf{no} then it will only show pointers that could not be reached, thus the errors.

\textbf{PERSONAL NOTE}: Valgrind tends to get complicated and thus I will not get into the depth of detail I did with GDB, if you think you can handle learning how to use it properly, it will be a useful tool throughout your coding career. I suggest reading up more on its uses and how it works in your free time.
Logical debugging

When dealing with things involving variables, calculation, numbers and other such functions, we tend to end up at the wrong result very often. These can be attributed to things such as off by one errors (Where one may have looped to one increment more than they intended) or accidental changing of a variable we thought may have been something else.

While I won’t go too into detail for this section, I would just like to mention what someone should do in these sort of situations.

When a result ends up completely different from what you expected, the best thing to do is to start from the very first thing in the calculation giving the error. Run a cout on the very first variable that will be affected by this function or calculation. Continue to move up or down the cout statement as necessary to check the new value of this variable and every other variable affected after every statement or line. More often than not you realise you may have used an incorrect data type, and you receive an incorrect value such as overflow from not using a long when the number is too big or losing the decimals when trying to store floating point in an integer. Follow the logic along yourself using a calculator and paper if required and see at exactly which point this will differ to what you expect.

Often code we think is completely correct ends up having one slight miscalculation that could potentially ruin the entire result of your program.

Do not be afraid to take out an exam pad or piece of paper, or even open an extra document in order to plan your program and follow along with the logic as you code. All of you should be doing COS 212 next year and you will realise almost all the programs you will do involve using around 20 to 30 pages in order to get it correct, especially when it comes to data structures which you will be dealing with later in this course.