Lecture 06
Debugging Programs with GDB and memory leaks

In this lecture

- What is debugging
- Most Common Type of errors
- Process of debugging
- Checking for memory leaks with valgrind
- Examples
- Further readings
- Exercises

What is Debugging

Debugging is the process of finding compile **time** and **run time errors** in the code. Compile time errors occur due to misuse of C language constructs. Most of the compile time errors can be overcome with careful attention to language syntax and use. Finding syntax errors is the first step in a debugging process. A typical C project that contains main.c, lib.h and lib.c can be compiled at once with

% gcc –ansi –Wall –pedantic lib.c main.c –o exec

It is also good to compile all source code files separately, so programmer can deal with errors more locally. For example, the main.c and lib.c in the above example can be compiled separately with –c flag.

% gcc –c –ansi –Wall –pedantic main.c

% gcc –c –ansi –Wall –pedantic lib.c

In this case, no executable is created and two files named main.o and lib.o will be created if source code is successfully compiled. Later on, one can create the executable by

% gcc lib.o main.o –o exec

Often dealing with syntax errors is the easiest part of debugging. Once you remove common errors like, missing semicolons, variable type mismatches, incorrect passing of

arguments, undefined or out of scope variables etc we can begin to execute the code.

Run time errors are the hardest to deal with. Run time errors can cause program to crash, or give you the incorrect output. First step in dealing with run time errors is to remove reasons for program to crash. Some of the most common errors of program crash are, dereferencing memory that has not been allocated, freeing memory that was already freed once, opening a file that does not exists, reading incorrect or non-existent data or processing command line arguments that have not been provided.

Although debugging a program is an art, it is important to develop a systematic process to debug. Often the best way to deal with errors is to not to introduce them in the first place. Developing individual functions and testing them to make sure they perform as expected is quite important. For each function, you provide the input and test for the expected output. You also need to deal with edge cases as they are most often the cause of the problem. Also it is important to test the input files to make sure that they are valid. One cannot assume that all input files are valid. For example, an input file may have a null line at the end of the file causing the program to behave in an unexpected manner. In the next section we will discuss some of the most common type of syntax and run time errors.

Most Common Type of Errors

There are no hard and fast rules about debugging code. But some errors occur more frequently than the others. We will start with a list of syntax errors that occur more frequently. Here is some of the most common type of errors.

- **Missing/misplaced semicolons**
  The most common type of error in the early stages of development. Each statement in a program must be separated by a semi colon. But avoid putting a semi-colon at the end of a loop declaration. This error can be corrected easily as compiler will provide enough clues as to the line where semi colons are missing or misplaced

- **variable type mismatches**

  C is a strongly typed language. We need to avoid mixing types. For example, a double cannot be assigned to an int or pointer should not be assigned to int etc. It is a good practice to type cast variables when necessary. For example, malloc always returns a void* but it can be typecasted to a int* or double* etc.

- **incorrect passing of arguments**

  pay special attention to function signature and how it is called. Often incorrect arguments are passed into the function. The danger in some cases is that compiler may not flag the errors as a compile error hence causing programs to fail at run time.

- **undefined or out of scope variables**

  C variables can be referenced within the scope in which they are declared. Typically compiler will give a detailed error message that can easily be recognized.

Run time errors are much more serious. There are many errors to look out for. Among some of them are

- **Array index out of bounds**

  This error is occurred when an out of bounds array index is referenced. In most cases a user can unintentionally assign A[n] = something to an array of size n. But problem comes when A[n] is dereferenced.

- **dereferencing illegal memory locations**

  Dereferencing memory locations that are set to NULL or simply invalid will cause programs to crash. It is always a good idea to test if a location is not NULL before dereferencing. It is also important to set pointers to NULL, when no memory is allocated for them. It is a good practice to assign a pointer to NULL after calling free.

- **Illegal use of pointers (*, **, ***)**
  In C code, we write a lot of code using *, ** and *** notations. While all of them are simply pointers, where they point to vary depending of the type. For example, int* is a pointer to an int, and int** is a pointer to an int*, or int*** is a pointer to an int** . Special attention must be paid when dealing with pointers.

- **missing command line arguments**
  Another common error is missing command line arguments. It is important that your program check for the number of command line arguments available using argc. If the number is less than the expected number, then provide a usage error message.

- **accessing null pointers (addresses)**
  NULL pointers are addresses that do not point to any location. For example, we may attempt to open a file using fopen, but if the file does not exists, then a NULL pointer is returned. Then when performing fscanf would cause a problem. Also it is important to check for NULL after calling malloc. Malloc may fail to provide the block of memory your requested, if it is not available for whatever the reason.

- **Invalid passing of arguments to functions**
  When passing arguments to functions, a great care must be taken to avoid invalid passes. For example, it is possible that one can pass an int to a pointer (no compile error) but trying to dereference the int would cause problems.

Here is the most common set of errors that we may encounter in this course.

A) dereference of uninitialized or otherwise invalid pointer
B) insufficient (or none) allocated storage for operation
C) storage used after free
D) allocation freed repeatedly
E) free of unallocated or potentially storage
F) free of stack space
G) return, directly or via argument, of pointer to local variable
H) dereference of wrong type
I) assignment of incompatible types
J) program logic confuses pointer and referenced type
K) incorrect use of pointer arithmetic
L) array index out of bounds

We will discuss this more in our lectures.

Process of Debugging

While most errors can be detected using manual debugging (eg: inserting printf statements), some errors are hard to detect just by observing the values of variables. The compiler may rearrange the order of the execution making the programmer think that an error occurred in a different place than the actual location. The best way to debug C code is to use some of the tools available from GNU. The most widely used tool for debugging C programs is the GDB.

GDB stands for GNU Debugger, is a powerful text debugger that will let you do many things. For example, you can stop program at specified location and see what has happened when program stopped. We can look at the values of the variables and Change things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another.

Typically when a program is compiled and run:

```
> gcc example1.c –o example1
> ./example1
Segmentation fault

> Why?!
Why?!: Command not found
```

When it is not clear why a program crashed, we can
Compile the program using gcc with the –g flag

> gcc myprogram.c -ggdb -o myprogram

Note that -ggdb flag allows source code to be compiled for the debugger. To start GDB

> gdb ./myprogram

There are number of gdb commands that are useful for a debugging session.

r(un) [arglist]
Runs your program in GDB with optional argument list

b(reak) [file:]function/line
Puts a breakpoint in that will stop your program when it is reached

c(ontinue)
Resumes execution of your program after it is stopped

n(ext)
When stopped, runs the next line of code, stepping over functions

s(tep)
When stopped, runs the next line of code, stepping into functions

q(uit)
Exits GDB

print expr
Prints out the given expression

display var
Displays the given variable at every step of execution

l(ist)
Lists source code

help [command]
Gives you help with a specified command

bt
Gives a backtrace (Lists the call stack with variables passed in)

Detecting Memory leaks

There are tools that detects and reports memory leaks. The most widely used tool is called “valgrind”. The Valgrind home page is at http://www.valgrind.org and you can find many resources on Valgrind there. To learn more about valgrind on unix, type

```
% man valgrind
```

NAME
valgrind - a suite of tools for debugging and profiling programs SYNOPSIS
valgrind [valgrind options] your-program [your-program options]
DESCRIPTION
valgrind is a flexible program for debugging and profiling Linux executables. It consists of a core, which provides a synthetic CPU in software, and a series of "tools", each of which is a debugging or profiling tool. The architecture is modular, so that new tools can be created easily and without disturbing the existing structure. This manual page covers only basic usage and options. Please see the HTML documentation for more comprehensive information.
INVOCATION
valgrind is typically invoked as follows:
valgrind program args
This runs program (with arguments args) under valgrind using the memcheck tool. memcheck performs a range of memory-checking functions, including detecting accesses to uninitialized memory, misuse of allocated memory (double frees, access after free, etc.) and detecting memory leaks.
To use a different tool, use the --tool option:
valgrind --tool=toolname program args
and more…….
To use Valgrind on Andrew unix, compile your code under

```
% gcc -g -ansi -pedantic -W -Wall main.c
```

Then run the code with Valgrind as

% valgrind --tool=memcheck --leak-check=full ./a.out

In addition to memcheck, valgrind has many other tools to check the use of functions, cache events etc. For now, we are only interested in making sure our programs don’t leak memory. The report provided by valgrind after running your code may look like

```
==18768== Memcheck, a memory error detector.
==18768== Copyright (C) 2002-2005, and GNU GPL'd, by Julian Seward et al.
==18768== Using LibVEX rev 1471, a library for dynamic binary translation.
==18768== Copyright (C) 2004-2005, and GNU GPL'd, by OpenWorks LLP.
==18768== Using valgrind-3.1.0, a dynamic binary instrumentation framework.
==18768== Copyright (C) 2000-2005, and GNU GPL'd, by Julian Seward et al.
==18768== For more details, rerun with: -v
```

```
==18768==
==18768==
==18768== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 4 from 1)
==18768== malloc/free: in use at exit: 30 bytes in 1 blocks.
==18768== malloc/free: 1 allocs, 0 frees, 30 bytes allocated.
==18768== For counts of detected errors, rerun with: -v
==18768== searching for pointers to 1 not-freed blocks.
==18768== checked 63,760 bytes.
==18768==
==18768== 30 bytes in 1 blocks are definitely lost in loss record 1 of 1
==18768== at 0x4905599: malloc (vg_replace_malloc.c:149)
==18768== by 0x400565: main (valgrind1.c:12)
==18768==
==18768== LEAK SUMMARY:
==18768== definitely lost: 30 bytes in 1 blocks.
==18768== possibly lost: 0 bytes in 0 blocks.
==18768== still reachable: 0 bytes in 0 blocks.
==18768== suppressed: 0 bytes in 0 blocks.
```

For now, we are interested in bytes that are definitely lost. Possibly lost bytes may be funny things that you may do with pointers such pointing to the middle of a heaped block etc. In all programs you write, you should look for memory leaks. Still reachable memory may be due to unreleased file pointers etc.

Examples

Example 1

Consider the following program

```
int main(int argc, char* argv[]){
   int x;
   printf("Please enter an integer : ");
   scanf("%d",x);
   printf("the integer entered was %d \n", x);
   return EXIT_SUCCESS;
}
```

The purpose of the program is to read an integer and print it out. But the program seg faults. Use gdb to find the error.

Example 2

Consider the following program

```
int main(int argc, char* argv[]){
   FILE* fp = fopen(“argv[1]”, “r”);
   char* word;
   while (fscanf(fp,”%s”,word)>0)
      {  }
   return 0;
}
```

The purpose of the program is to read a set of strings from a file. But the program seg faults. Use gdb to find the error.

Example 3
Consider the following program

```
int main(int argc, char* argv[]){
    printf("%ld \n", INT_MAX);
    int n = INT_MAX;
    int A[n];
    int i = 0;
    while (i<n)
      A[i] = rand()%10;

   return EXIT_SUCCESS;
}
```

The purpose of the program is to allocate an array of ints and initialize them to a random number. But the program seg faults. Use gdb to find the error.

Further Readings
[1] K & R

Exercises
[1] Write a program that reads a file of strings, allocate memory for each string dynamically, and store the strings in an array of char*'s. Calculate (using sizeof operator) memory usage by your program.

[2] Write code to free all dynamic memory allocated in the above case. Check with valgrind to make sure memory is deallocated.

Exercises

1. Consider the following program,

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

int main(int argc, char* argv[]){
  char* name = malloc(strlen(argv[1])) ;
  name = strcpy(name,argv[1]);
  printf("%s \n", name);
  return EXIT_SUCCESS;
}
```

This code may throw a segmentation fault. Use debugger to find out where the errors are and fix them.

2. Consider the following program. Use the GDB to find the errors

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

int main(int argc, char* argv[]){
       int* ptr = malloc(4);
       free(ptr);
       scanf("%d",*ptr);
       return EXIT_SUCCESS;
}
```

3. Consider the following code. Explain the error w/o using GDB

```
int main(int argc, char* argv[]){
  int** A;
  foo(&A);
  return 0;
}

foo(int*** array){
    int** arrayint = (int**)malloc(2*n*sizeof(int*));
    for (i=0;i<n;i++)
       arrayint[i] = (*array)[i];
    free(*array);
    array = &arrayint;
   }
```

4. What is wrong with the following code

```
int main(int argc, char* argv[]){
      int i=0,n=1000;
      char name[32];
      char* A[n];
      FILE* infile = fopen(argv[1],”r”);
      while (fscanf(infile,"%s", name) > 0)
          strcpy(A[i++],name);

/*shift all elements to right by 1*/
 for (j=i; j>0; j--)
        A[i+1] = A[i] ;
```