ECE 244   **Programming Fundamentals**

Lab Assignment #1: The ECF Development Environment

1   Objectives

The objectives of this assignment are multifold: to learn basic UNIX commands, to learn how to submit your solutions to the `autotester` program, and to provide you with practice on the edit-compile-run-debug cycle of C++ programs in a UNIX environment.

2   Problem Statement

In this lab you will first create a directory for ECE244 assignments in the file system under your UNIX account on ECF. You will then copy a simple C++ program, compile it, check it for correct output, edit the program, recompile and submit to the autotester. Second, you will use a debugger called `DDD` to debug five programs. A step-by-step tutorial is given for debugging the first three programs to get you familiar with the basic functions of the debugger. You debug the other two programs on your own.

3   Background

3.1   Logging into ECF

Before you can gain access to the ECF computer system, you must have your *login name* and a *password* to log onto the system. Your login name (or just *login*) is a 6-8 character string that is used to identify you to the computer system. Your password is an 8 character string known only to you and is used to provide security for your computer account. No one knows what your password is except you, not even the ECF system! **<u>You must never share your password with anyone</u>**.

Once you log in, you will likely be presented with some introductory messages and announcements from ECF, and then you will get an *xterm* window. If you do not get the xterm window you can start one by clicking on "Applications" then "System Tools", and then "Terminal". The xterm window has the UNIX *prompt*: "p*xxx*.ecf%" (e.g., p120.ecf% or p150.ecf %) indicating that the system is ready to accept your commands. In the remainder of this document, we will use just "%" as the prompt and it will appear in front of all commands you type, but of course you will not type it.

3.2   The UNIX File System

3.2.1 The File Structure

The UNIX file system is organized into a tree-like structure of directories. At the top of the tree is the <u>*root*</u> directory of the system, designated by a "/". Various system directories, named "bin", "local", "lib", "u", etc are contained in the root directory. In turn, files and other directories exist in each of these directories, as shown in Figure 1.

Your files and directories exist in the system under your *home directory*, which has the same name as your login name (assumed to be `ast` for the remainder of this document). This is also shown in Figure 1: your files are in the directory called `ast`, which exists in the directory called `1T0`, which in turn is in the directory called `u` that exists in `/`.
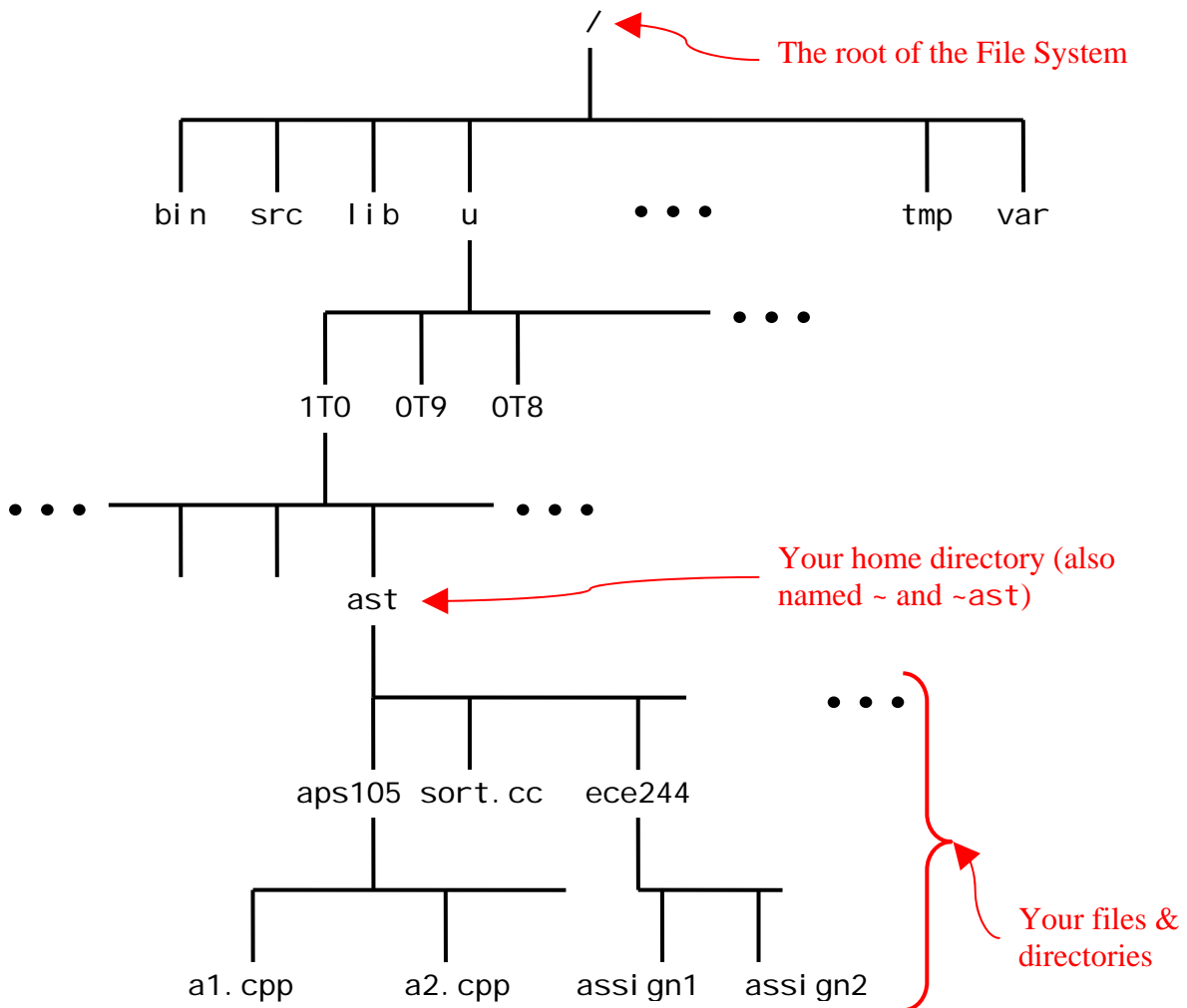

Figure 1: An example UNIX directory structure.

3.2.2 The Working Directory

When you are logged into the ECF system, you are always associated with one directory or another. The directory you are currently associated with is called your *working directory*. When you first login, your home directory is your working directory. You can then change your working directory using the `cd` command (explained later). At any time, you can find out what your working directory is using the print-working-directory command, `pwd`, as follows:

```
% pwd
```

The command will print the name of your current working directory to the screen.

3.2.3 File/Directory Locations

The location of a file or a directory is specified using a _path_ through the tree. This path can be _full_ (or _absolute_) starting from the root of the tree "/", or _relative_ to the current directory, or to some other directory. Thus, in Figure 1, the file a1.cpp can be specified in a number of ways, assuming that your home directory is your working directory:

/u/1T0/ast/aps105/a1.cpp	full or absolute
aps105/a1.cpp	relative to the working directory
~ast/aps105/a1.cpp	relative to the home directory of ast
~/aps105/a1.cpp	relative to your home directory

Note how ~ast specifies the home directory of user "ast" and how "~" by itself specifies your home directory.

UNIX also provides two short-hands: ". " refers to the working directory, and ". . " refers to the parent directory of the working directory. These short-hands are useful in specifying relative paths. For example, in Figure 1, if the working directory is aps105, then the file a1.cpp can also be specified as ./a1.cpp, and the file sort.cc can be specified as ../sort.cc.

3.2.4 Listing Contents of a Directory

To obtain a listing of the names of all of the files in a directory, you can use the ls command:

    % ls

Thus, using ls when you first login will print a list of all the files in your home directory.

3.2.5 Creating a Directory

In order to keep your files better-organized, you should create directories to hold files relating to various subjects, courses, etc. For example, to create a new directory, called newdir, in your home directory, use the mkdir command at the prompt:

    % mkdir newdir

Since newdir is contained in your home directory, newdir is referred to as a _subdirectory_ of your home directory, and your home directory is referred to as the _parent directory_ of newdir. Similarly in Figure 1, aps105 is the parent directory of the file a2.cpp while ece244 is a subdirectory of the home directory ast.

3.2.6 Changing the Working Directory

To work within a directory, you must use the change-directory command, cd, to make it your working directory. The cd command has the following syntax:

    cd dirlocation

where `dirlocation` is the location of the directory you wish to make your working directory, specified using a path in the directory tree as described above. For example, in Figure 1, if your home directory is the working directory, and you can make `assign1` your working directory, by typing:

```
% cd ece244/assign1
```

You may also accomplish the same thing by specifying the full path to the directory as follows:

```
% cd /u/1T0/ast/ece244/assign1
```

If your working directory is `assign1`, and you can make `ece244` your working directory by typing:

```
% cd ..
```

which takes you to the parent directory of your current directory; i.e., `ece244`.

Finally, you can always return to your home directory by typing:

```
% cd
```

with no directory location specified.

It is important to always know what your current working directory is (using the `pwd` command, described earlier), and how to move to subdirectories of it, or to move to its parent directory.

3.3 Moving, Copying and Deleting Files

Using UNIX commands, you can type commands to list your files, move, copy, delete, etc. For example, the `cp` command is used to create a copy (duplicate) of a file. It has the following syntax:

```
cp file1 file2
```

This takes a file named `file1` and creates a new file called `file2` that is an exact duplicate of `file1`.

The `mv` command is used to move (or rename) a file, and has the following syntax:

```
mv file1 file2
```

This takes a file named `file1` and creates a new file called `file2` that is an exact duplicate of `file1`. The original `file1` is removed.

Finally, the `rm` command is used to delete (or remove) a file:

```
rm file
```

Be careful when using the `rm` command; files that you remove cannot be retrieved. Thus, it pays to use the `rm` command with the "`-i`" option as follows:

```
rm –i file
```

The command will prompt for confirmation before the file is deleted.

3.4   Protecting Your Files

Every UNIX file has an owner (and group) associated with it. You are the owner of all files and directories in your home directory. As the owner you may choose to allow or deny others to view your files. In order to protect your assignments from being copied by others, you should use the chmod command to protect you files. When you create a new directory newdir, use the chmod command as follows (assuming your working directory is the parent of newdir):

```
% chmod –R go-rwx newdir
```

This prevents others from reading or searching through newdir. You can use the command on any existing directory as well to protect it and all the files contained in it.

3.5   Getting More Information on UNIX Commands

You can get more information on any UNIX command by using the man command (for manual page). For example, type

```
% man ls
```

to get information on the ls command. As the information is displayed, --More-- appears at the bottom of the screen. At this point, you can press the spacebar to display the next screen, or press q to quit.

Indeed, if you wish to find out more about the man command, type:

```
% man man
```

3.5   Debuggers

A *debugger* is a tool that helps you find errors (i.e., "bugs") in your programs. It does so by allowing you to control the execution of a program. You can make the program stop when it reaches a particular point in the code: this is called a *breakpoint*. While at a breakpoint, you can examine the values of variables and inspect the state of the program. The source code of the program is displayed in a window. You can *resume* execution from the breakpoint, possibly to another breakpoint, or *single-step* one statement at a time in your program. You can also set *watches* to view the values of variables you are interested in.

The use of a debugger can save you hours of frustration. While a debugger can never tell you what a bug is, it can help you quickly determine where the bug is located in your code. Combined with your knowledge of what the program should be doing, you can quickly locate and fix your bug.

Learning to use a debugger does take some time. However, this is time well-spent. It certainly beats debugging your code by staring at it for hours at a time, by randomly changing lines in the code, or by inserting print statements all over the code.

This lab is designed to help you learn how to use a debugger called Data Display Debugger (DDD). It is actually a graphical front-end to a powerful debugger called the Gnu Debugger (GDB). If you are familiar with another debugger, feel free to use it instead of DDD.

This lab assumes that you know how to use a text editor to edit your programs. Examples of such editors include emacs, vi, and nedit.

4   Preparation

Please read the previous section on UNIX file system and commands. Also, please read the Laboratory information handout and any material posted to the Lab section of the course's web site on the same topic. Finally, please familiarize yourself with the submitece244f, exercise, and chksubmit commands. Full documentation of these commands can be found at: http://www.ecf.utoronto.ca/~stumm/ece244.html.

It is vital that you go through all the steps of this assignment, even when you think it is too easy to do. The assignment will prepare you for the process of checking and submitting your assignment. Just as incentive, keep in mind that last year as many as 20% of the class received a mark of 0 for an assignment, not because they did not work on it (indeed they put more than 40-50 hours of work), but because they failed to use the submit commands to properly submit their assignments.

There is a tutorial on how to use the DDD debugger, as well as the manual of DDD, in the "Labs" section of the course's web. Read through the tutorial and the first few sections of the manual before the lab. **DO NOT PRINT THE DDD MANUAL!** It is too long (220 pages!) and is probably not useful to you at this stage.

5.   Procedure

5.1   Directory preparation

Use your login and password to log onto the ECF system.

In your home directory, create a new directory for ECE244 (this course!) by typing the following at the prompt:

```
% mkdir ece244
```

Make sure that the directory is protected by typing:

```
% chmod go-rwx ece244
```

Now use the ls command to see the new "ece244" directory in your list of files.

Change your working directory to ece244 using the following command:

```
cd ece244
```

If you type the `ls` commend now, you will get no output, since the directory is currently empty; i.e., has no files in it.

Now create a directory called `assign1` in ece244 and make it your working directory.

5.2 The "`Hello, World!`" Program

You will use a simple example program whose only purpose is to print the message "`Hello, world!`" to the screen. An initial version of this program is provided; you simply have to copy it into your working directory (`ece244/assign1`). You can do so by executing the following command:

> `% cp ~stumm/public/ece244/lab1_submit/hello.cpp .`

This command copies the file called `hello.cpp` from the directory "`public/ece244/lab1_submit/`" in the home directory of user "`stumm`" into your current working directory.

Use the `ls` command to ensure that the file exists

Compile the program by executing the command:

> `% g++ hello.cpp –o hello`

This will create an executable called `hello`. You can run it with by typing:

> `% ./hello`

which runs the executable called `hello` in the working directory.

5.3 Testing the "`Hello, World!`" Program

The automarker program will test all code that you submit. It does so by compiling your programs and then running them with various inputs, or *test cases*. Each time it runs your program it compares the output to that of the reference solution. If your output matches the reference output then your program passes the test case, otherwise it fails. In general this works quite well – if your program does what it's supposed to do, you will pass all the test cases, and if it doesn't then you will fail some of them. Unfortunately there are two things that could go wrong if you're not careful.

First, the automarker requires that your program have a very specific name. This name will always be specified in the lab handout, and if your program's name differs even slightly then the automarker will not be able to run it. This will mean that *__you will fail every test case__*, even if your program works! Fortunately, we have provided a program that double-checks that you have named your program correctly. It is called `exercise`, and it has a subset of the test cases that will be used by the automarker. It chooses one at random and uses it to test your program. If your output does not match the expected output `exercise` will tell you. The `exercise` command has the following syntax:

```
~stumm/public/ece244/exercise assignment# program_name
```

Let's assume that our example program should be named `Hello`. But, as things stand, an executable called `hello` (note the lowercase "h") is created when you compile it. If you didn't notice this and submitted the files as-is you would fail every test case. If you run exercise, however, it would warn you of this problem:

```
% g++ hello.cpp -o hello
% ~stumm/public/ece244/exercise 1 ./hello

Error - exercise only supports the following executables: Hello
Quitting!
```

If `exercise` is run with an executable that does not match the name expected by the automarker, it will print a message like the one above (note that the error messages may be slightly different from the ones printed in this document). To fix the problem, _**copy**_ `hello.cpp` into another file `Hello.cpp` using the `cp` command, and re-compile:

```
% g++ Hello.cpp -o Hello
```

Now run exercise again:

```
% ~stumm/public/ece244/exercise 1 ./Hello

Running the following input on your program:
==========================================
------------------------------------------
Your program produced the following output:
==========================================
Hello world!
------------------------------------------
Expected to see the following output:
==========================================
Hello, world!
------------------------------------------

Unacceptable output produced.
```

Note that `exercise` no longer prints an error message regarding the file name. It does, however, print a different error message. This is an example of the second major pitfall of the automarker: typos. Any difference from the expected output, no matter how minor, will cause your program to fail a test case. Here the program omitted a comma in its output.

Now fix the problem in the example program. Edit `Hello.cpp` and add the missing comma on line 5. The result should look like this:

```
cout << "Hello, world!\n";
```

Now recompile and run `exercise` one more time:

```
% g++ Hello.cpp -o Hello
% ~stumm/public/ece244/exercise 1 ./Hello

Running the following input on your program:

==========================================
------------------------------------------
Your program produced the following output:
==========================================
Hello, world!
------------------------------------------
Expected to see the following output:
==========================================
Hello, world!
------------------------------------------

Acceptable output produced.
```

This time `exercise` tells you that the output was correct. If, unlike this simple example, your program outputs different things depending on the input, you should run `exercise` several times. This will test your program against all the different test cases that are available to `exercise`. You should use `exercise` for every assignment, but you should also remember that passing all of its test cases does not guarantee a perfect mark – the automarker uses some test cases that are not available to `exercise`. You should always perform additional testing.

6   Using the DDD Debugger

7.1 Part I – Your First DDD Session

Often, a C++ program will terminate execution because of a run-time error, giving a "bus error" or a "Segmentation fault" error message, which is not very helpful. One of the most common uses of a debugger is to determine where the error occurred and what caused it. In this part of the assignment, you will use the DDD debugger and a sample program to learn how to determine where such a run-time error occurred.

Use a browser to download the C++ program called `list.cc` from the "Labs" section of the course's web site. The program creates three nodes and links them in a linked list, as was explained in class. Look at the program and study it.

Compile this program using Gnu's C++ compiler (called g++) as follows:

```
g++ list.cc –g –o list
```

The `–g` option tells the compiler to compile the program for debugging. **This option is required when compiling the program, or else you will not be able to use the debugger to debug the program.** The `–o` option tells the compiler to place the executable in the file called `list`. Thus, the program may be executed by typing:
```
./list
```

The program ends with the error message: `Segmentation fault`.
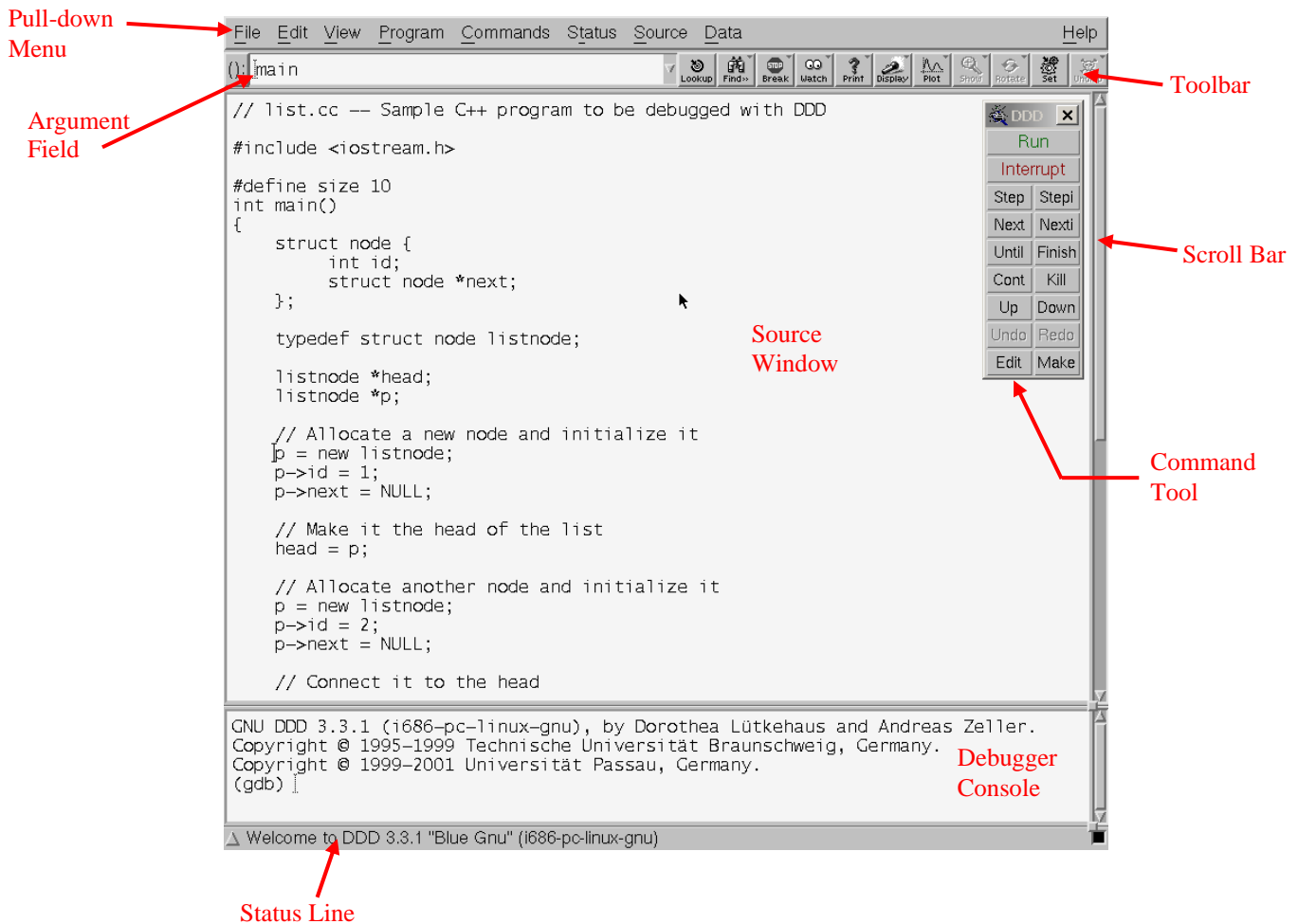
Start the debugger by typing:

```
ddd list
```

After a few seconds, DDD comes up. The initial DDD window is shown in the screenshot below. The "Source Window" contains the source of list.cc. You can use the scroll bar to scroll through the code.

The top menu bar contains a number of pull-down menu items that allow you to open, close, and edit files, to add and remove views, and to execute commands. Under this top menu is a menu bar with several icons on it. You can move you mouse over an icon and leave it there for a short time to see a short description of what it does. The icons deal with text lookups (searches), breakpoints, and data viewing. Some of these operations require that an argument be given to the corresponding command. This can be done through the "Argument field" box to the left of the icons. For example, to display the value of a variable, the name of the variable is specified in the "Argument field".

The "Command Tool" contains buttons that correspond to frequently used commands. The buttons allow you to start and stop the execution of a program, single-step through the program, as well as edit and recompile the program.

Underneath the "Source Window" is the "Debugger Console" window, or simply the GDB console. This window displays your interaction with the debugger and with your program. It is used to input commands to the debugger at the GDB prompt "(gdb)". Remember, DDD is a graphical interface to the GDB debugger.

Finally, at the very bottom of the DDD window, is the "Status Line". It displays helpful messages on what DDD is doing, as well as useful information when you move the mouse around.

```
// list.cc -- Sample C++ program to be debugged with DDD

#include <iostream.h>

#define size 10
int main()
{
    struct node {
        int id;
        struct node *next;
    };

    typedef struct node listnode;

    listnode *head;
    listnode *p;

    // Allocate a new node and initialize it
    p = new listnode;
    p->id = 1;
    p->next = NULL;

    // Make it the head of the list
    head = p;

    // Allocate another node and initialize it
    p = new listnode;
    p->id = 2;
    p->next = NULL;

    // Connect it to the head
```

```
GNU DDD 3.3.1 (i686-pc-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" (i686-pc-linux-gnu)

You can run the program in the debugger in one of two ways. You can click on the "Run" button in the "Command Tool", or you can use the "Program" pull-down menu and select the "Run" option. The latter choice allows you to also provide run-time arguments to your program, but this is not needed in our case.

Click the "Run" button in the "Command Tool". The program will start to execute and then stops with a "Segmentation fault" as shown in the screenshot below. The arrow indicates where the error occurred, at line 40: p->next->next = p. This is certainly much better than spending hours staring at your code, or trying to guess where the error occurred by inserting print statements all over the code!
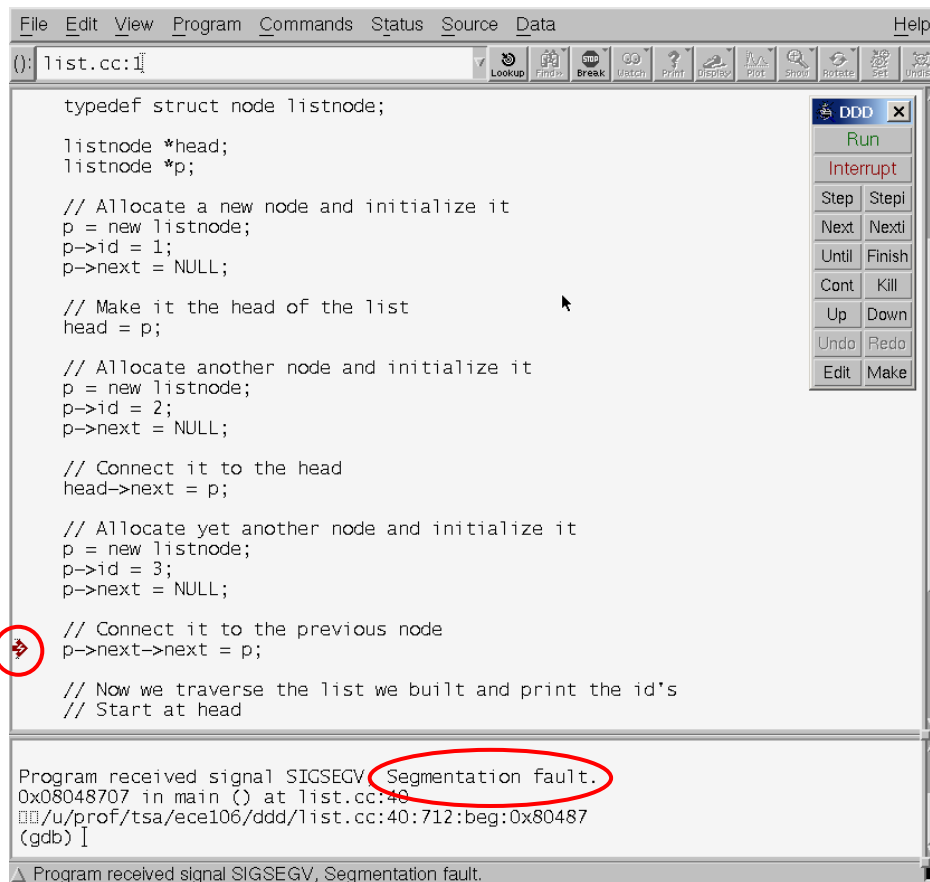
Now, you must find out what caused the "Segmentation fault". You can do so by examining the values on both the left and right hand sides of the assignment at line 40. You can find out the value a variable has when the program is stopped in one of two ways (there are actually more, but let's stick with the most common two). The first is to use *value hints*. Highlight the name of a variable and move the mouse cursor over the highlighted text and leave there for a moment. A small window will appear that shows the type and value of the variable. The status line also shows the value of the variable.

Highlight p on the right hand side of the assignment to find out its value. The result is shown in the screenshot below.
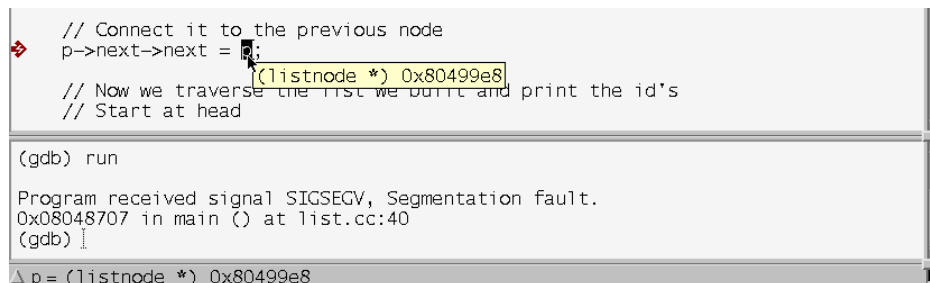
The second way to find the value of a variable is to use the *print value* command to print the value of the variable to the GDB console. You can do this by copying the name of the variable to the "Argument field" box and then clicking on the "Print" icon on the toolbar. You can also do this by highlighting the variable and clicking the right button of the mouse for a popup menu. Select the Print option. Try both. A screenshot of the popup menu is shown below.

```
     // Connect it to the previous node
  ⇨  p->next->next = p:
                    Print p
     // Now travers Display p  st we built and print the id's
     // Start at head
                    Print *p
─────────────────── Display *p ──────────────────────────
(gdb) run           What is p   GV, Segmentation fault.
Program received sign           .cc:40
0x08048707 in main (: Lookup p
(gdb)               Break at p
△ Print the value of the selec Clear at p  he GDB console
                              Clear at p
```

Back to the "Segmentation fault". Find the value of the left hand side of the assignment at line 40: `p->next->next`. DDD indicates that it is not possible to access memory at this address, as shown in the screenshot below. This is the source of the error (the word "segment" refers to an area in memory and a program accessing memory where it should not or cannot causes a "segment"-ation fault).

```
     // Connect it to the previous node
  ⇨  p->next->next = p;
              (node *) Cannot access memory at address 0x4
     // Now we traverse the list we built and print the id's
     // Start at head
─────────────────────────────────────────────────────────
(gdb) run

Program received signal SIGSEGV, Segmentation fault.
0x08048707 in main () at list.cc:40
(gdb)
△ p->next->next = (node *) Cannot access memory at address 0x4
```

Find the value of the variable `p->next` on the same line. DDD indicates that it is `0x0` (i.e. `NULL`), as shown in the screenshot below. This is what is causing the problem. The program is de-referencing a `NULL` pointer! The code is simple enough, and you know that `p->next->next` should really be `head->next->next`.

```
     // Connect it to the previous node
  ⇨  p->next->next = p;
       (node *) 0x0
     // Now we traverse the list we built and print the id's
     // Start at head
─────────────────────────────────────────────────────────
(gdb) run

Program received signal SIGSEGV, Segmentation fault.
0x08048707 in main () at list.cc:40
(gdb)
△ p->next = (node *) 0x0
```

Fix the problem by exiting the debugger and editing the file to make the correction (later you will learn how to do this while the debugger is running and/or from within the debugger). Recompile the program. It should now run correctly.

7.2 Part II – Breakpoints

It is also often the case that a program runs to completion but does not behave as it should. The debugger becomes handy in trying to figure out what is wrong with the program. This is usually done by controlling the execution of the program, inspecting the values of the variables, and comparing these values with what they are expected to be. A variable that takes a value it should not, according to how a program is expected to work, is often the first clue to determining what is wrong with the program.

Use your browser to download the program `bubble.cc` from the "Labs" section of the course's web site. The program prompts for 5 integers, stores them in an array, and then sorts the array using a bubble sort function. Look at the program and study it.

Compile this program using the `g++` compiler as follows:

```
g++ bubble.cc -g -o bubble
```

Again, the `-g` option tells the compiler to compile the program for debugging. The `-o` option tells the compiler to place the executable in the file called `bubble`. Thus, the program is executed by typing:
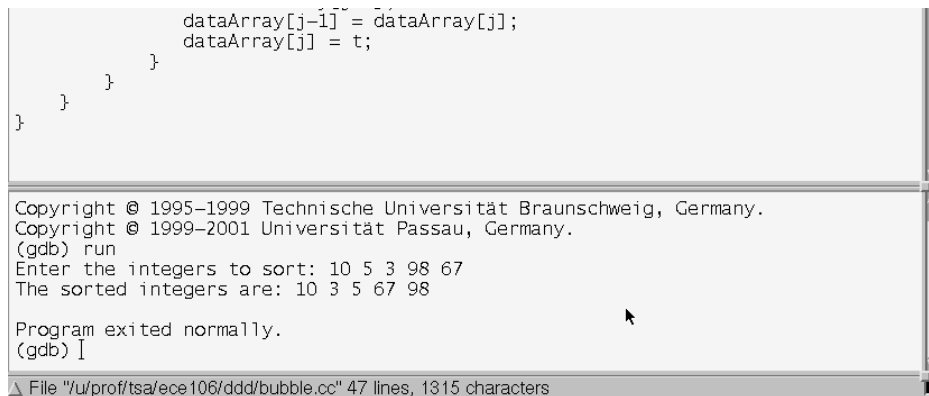
```
./bubble
```

The program runs to completion, but it does not sort properly. For example the input `10 5 3 98 67` produces the output `10 3 5 67 98`. While the last four elements are sorted, the value `10` is out of place!

Start the debugger by typing:

```
ddd bubble
```

Run the program by clicking on the "Run" button in the "Command Tool". The `GDB` console shows the output of the program, prompting the user for the 5 integers, as shown in the screenshot below. Enter `10 5 3 98 67` followed by `return`. The console window shows the incorrect output of the program. Note that you may have to click the mouse in the console window to interact with `GDB`.

```
                    dataArray[j-1] = dataArray[j];
                    dataArray[j] = t;
                }
            }
        }
}

Copyright © 1995-1999 Technische Universität Braunschweig, Germany.
Copyright © 1999-2001 Universität Passau, Germany.
(gdb) run
Enter the integers to sort: 10 5 3 98 67
The sorted integers are: 10 3 5 67 98

Program exited normally.
(gdb)
∆ File "/u/prof/tsa/ece106/ddd/bubble.cc" 47 lines, 1315 characters
```
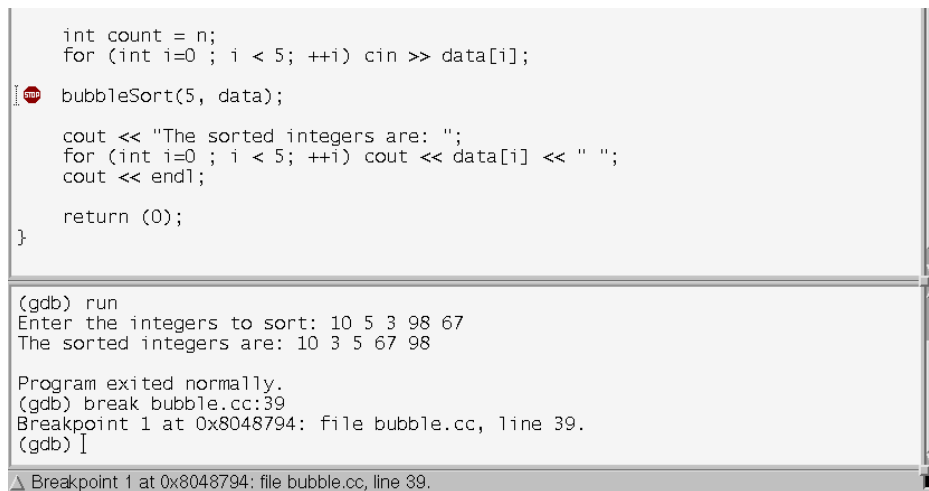
To determine what the problem is, start by inserting a *breakpoint* in the program. A breakpoint is used to stop the execution of the program at a point of your choice. You can then inspect the values variables have at this point. There are several types of breakpoints. In this lab we will focus on the most common one, a statement breakpoint, which allows you to stop execution at a given source statement (i.e., line).

The purpose of the use of the first breakpoint is to confirm that the array `data` does indeed contain what you expect it to contain.

Scroll the "Source Window" to get the main function of the program in view. Left-click the mouse at line 39: `bubbleSort(5, data)`. You should see the line number displayed in the "Argument field". Click on the icon with the red "STOP" sign and the work "Break" in the toolbar. This inserts a breakpoint **_before_** the call to `bubbleSort`. You can clearly see that a breakpoint has been inserted from the red "STOP" sign at line 39. You can also see this from the console window, which indicates that a breakpoint, numbered 1, has been inserted in file `bubble.cc` at line 39 (`0x8048794` is the address of breakpoint, which you may ignore). The screenshot below shows the DDD window after you take the above actions.


```
        int count = n;
        for (int i=0 ; i < 5; ++i) cin >> data[i];

   ⬤    bubbleSort(5, data);

        cout << "The sorted integers are: ";
        for (int i=0 ; i < 5; ++i) cout << data[i] << " ";
        cout << endl;

        return (0);
}
```

```
(gdb) run
Enter the integers to sort: 10 5 3 98 67
The sorted integers are: 10 3 5 67 98

Program exited normally.
(gdb) break bubble.cc:39
Breakpoint 1 at 0x8048794: file bubble.cc, line 39.
(gdb)
```

△ Breakpoint 1 at 0x8048794: file bubble.cc, line 39.

Now, you can run the program and re-input the same data. The program does not terminate, rather, it stops executing at line 39. This is before the function `bubbleSort` is called, and is indicated by the green arrow at the line to the left of the breakpoint sign. It is also indicated in the console window from the line `Breakpoint 1, main () at bubble.cc:39`. The line says that execution reached breakpoint 1, which is in the function `main` in the file `bubble.cc`. With the program stopped, you may examine the values in the array `data` as shown in the screenshot below. The displayed values confirm that `bubbleSort` is passed correct input.

```
        int count = n;
        for (int i=0 ; i < 5; ++i) cin >> data[i];

        bubbleSort(5, data);
                      {10, 5, 3, 98, 67}
        cout << "The sorted integers are.  ";
        for (int i=0 ; i < 5; ++i) cout << data[i] << " ";
        cout << endl;

        return (0);
}
```

```
Breakpoint 1 at 0x8048794: file bubble.cc, line 39.
(gdb) run
Enter the integers to sort: 10 5 3 98 67

Breakpoint 1, main () at bubble.cc:39
(gdb)
```

```
△ data = {10, 5, 3, 98, 67}
```

You may remove the breakpoint by right-clicking on the red "STOP" sign and choosing "Delete Breakpoint". For now, just leave it in. Scroll the "Source Window" to bring `bubbleSort` into view and insert another breakpoint before the first `for` loop. The purpose of this breakpoint is to stop execution before any of the function's code begins to execute and confirm that the array values are correct inside the function.

Continue the execution of the program by clicking the "Cont" button in the "Command Tool". This button continues the execution of the program from the first breakpoint to the next. Alternatively you may single-step execution one statement at a time by pressing the "Step" button until the breakpoint is reached. The program stops execution and stops before the `for` loop. You may then examine the values in `dataArray`, as shown in the screen shot below.

```
void bubbleSort(int n, int dataArray[]) {
    int t;
    for (int i = n; i >= 1; --i) {
        for (int j = 2; j <= i; ++j) {
            if (dataArray[j-1] > dataArray[j]) {
                t = dataArray[j-1];    (int *) 0xbfffe480
                dataArray[j-1] = dataArray[j];
                dataArray[j] = t;
            }
        }
    }
}
```

```
(gdb) break bubble.cc:17
Breakpoint 2 at 0x80486a7: file bubble.cc, line 17.
(gdb) cont

Breakpoint 2, bubbleSort (n=5, dataArray=0xbfffe480) at bubble.cc:17
(gdb)
```

```
△ dataArray = (int *) 0xbfffe480
```

In this case, the value hint indicates that `dataArray` is a pointer to an integer and gives its value. The reason is that arrays are passed to functions as pointers (more on this will be covered in the lectures). To display the array, you must use the "print" command from the GDB console using the `print` command as follows:

```
print dataArray[0]@5.
```

This tells the debugger to treat `dataArray` as an array, not a pointer, and to print 5 values starting from index 0. The output of the debugger is shown in the screenshot below and confirms that the values of the array elements are correct inside the function.


```
void bubbleSort(int n, int dataArray[]) {
    int t;
    for (int i = n; i >= 1; --i) {
        for (int j = 2; j <= i; ++j) {
            if (dataArray[j-1] > dataArray[j]) {
                t = dataArray[j-1];          (int *) 0xbfffe480
                dataArray[j-1] = dataArray[j];
                dataArray[j] = t;
            }
        }
    }
}
```

```
(gdb) cont

Breakpoint 2, bubbleSort (n=5, dataArray=0xbfffe480) at bubble.cc:17
(gdb) print dataArray[0]@5
$1 = {10, 5, 3, 98, 67}
(gdb)
```

△ dataArray = (int *) 0xbfffe480

Next, insert a break point before the `if` statement on line 19. This will allow you to stop execution on every iteration of the nested loops, and examine the values of the elements of the array, in hope of finding out the source of the bug. Continue the execution of the program. Execution will stop in the first iteration, which is supposed to compare the first and second elements of the array to determine which is bigger, and move the bigger element if necessary. Inspect the value of `j` and look at the array elements at indices `j-1` and `j`. The value of `j` is 2. Thus the program is comparing elements 1 and 2 in the first iteration. Since C++ arrays start at 0, these are the second and third elements, not the first and second. You have the cause of the bug! The `j`-loop is written assuming the array starts at index 1 and ends at index n, while C++ arrays start at index 0 and end at index n-1! Thus, the loop should be shifted by 1.


```
void bubbleSort(int n, int dataArray[]) {
    int t;
    for (int i = n; i >= 1; --i) {
        for (int j = 2; j <= i; ++j) {
            if (dataArray[j-1] > dataArray[j]) {
                t = dataArray[j-1];
                dataArray[j-1] = dataArray[j];
                dataArray[j] = t;
            }
        }
    }
}
```

```
(gdb) break bubble.cc:19
Breakpoint 3 at 0x80486d0: file bubble.cc, line 19.
(gdb) cont

Breakpoint 3, bubbleSort (n=5, dataArray=0xbfffe480) at bubble.cc:19
(gdb)
```

△ j = 2

Exit the debugger and edit the program to change the line `for (int j = 2; j <= i; ++j)` to `for (int j = 1; j  < i;  ++j)`, which should iterate correctly over the elements of the array. Re-compile the program. It should now run correctly.

7.3 Part III – The Data Window

In this part of the assignment, you will learn one of the most powerful features of DDD, that is, its graphical display. This feature allows you to display the names and values of variables in a "Data Window," which is updated each time your program's execution stops. The feature is particularly useful in debugging programs with large and complex data structures. In this part, you will also learn about how to update your program while the debugger is running.

Use a browser to download the C++ program called `list2.cc` from the "Labs" section of the course web site. The program creates three nodes and links them in a linked list, and then it creates a fourth node and inserts it between the second and third nodes in the list. The program is very similar to the one you looked at in **Part I** above. Look at the program and study it. It should print the "id's" of the nodes as: `1 2 4 3`.

Compile this program as follows:

```
g++ list2.cc –g –o list2
```

Again, the `–g` option tells the compiler to compile the program for debugging and the `–o` option tells the compiler to place the executable in the file called `list2`. The program may be executed by typing:
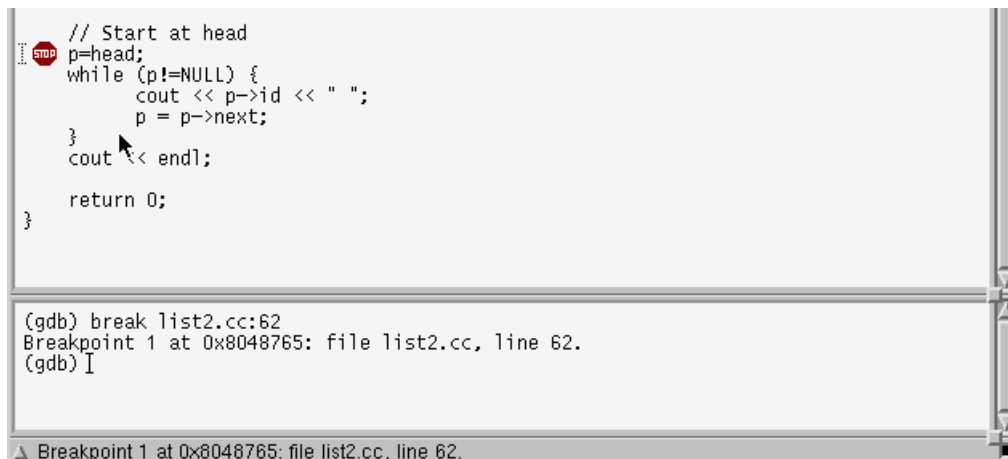
```
./list2
```

The program enters into an infinite loop, and you will see "4's" printed all over the screen. Terminate the program by repeatedly typing "`^C`" (press and hold the control key while repeatedly hitting the `C` key). The program will eventually terminate.

Start the debugger by typing:

```
ddd list2 &
```

The `&` at the end of the command line runs DDD in the "background," allowing you to type commands into your `xterm` window while the debugger is running.

There is only one "while" loop in the program, and thus, it must be the source of the infinite loop. Insert a breakpoint before this loop, as shown below.

```
        // Start at head
[STOP]  p=head;
        while (p!=NULL) {
                cout << p->id << " ";
                p = p->next;
        }
        cout << endl;

        return 0;
}
```
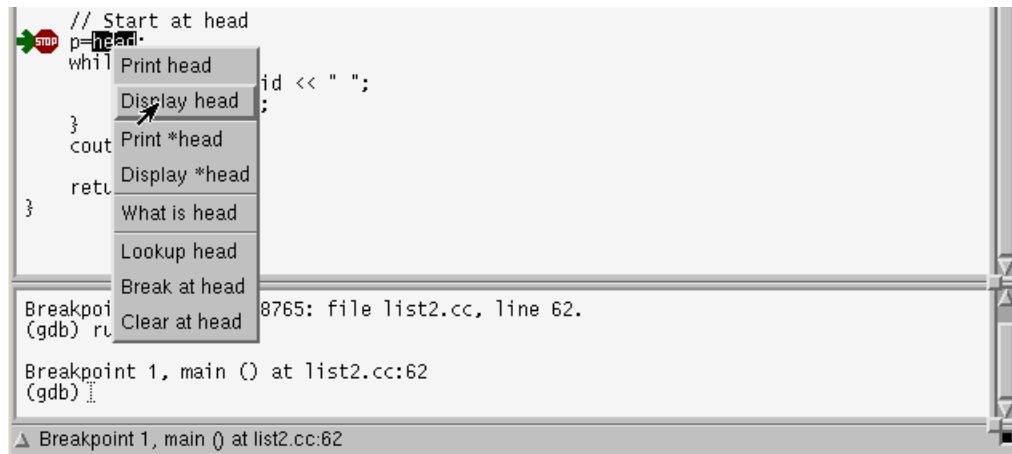
```
(gdb) break list2.cc:62
Breakpoint 1 at 0x8048765: file list2.cc, line 62.
(gdb)
```

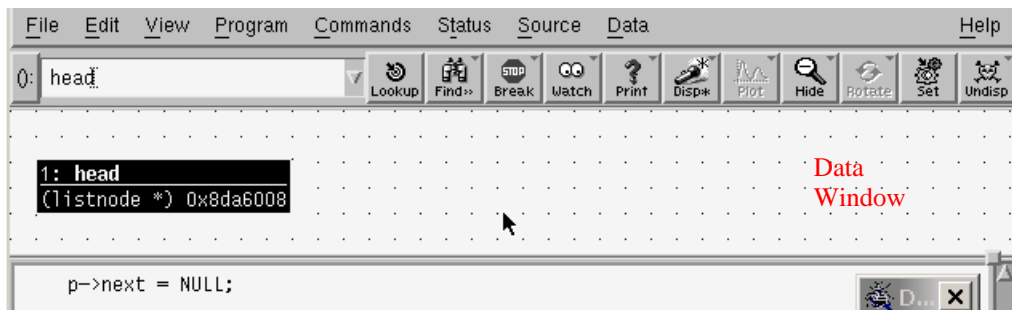Breakpoint 1 at 0x8048765: file list2.cc, line 62.

Now run the program by clicking on the "Run" button in the "Command Tool". The GDB console shows that the program is stopped at the breakpoint as shown below.

```
Breakpoint 1 at 0x8048765: file list2.cc, line 62.
(gdb) run

Breakpoint 1, main () at list2.cc:62
(gdb)
```
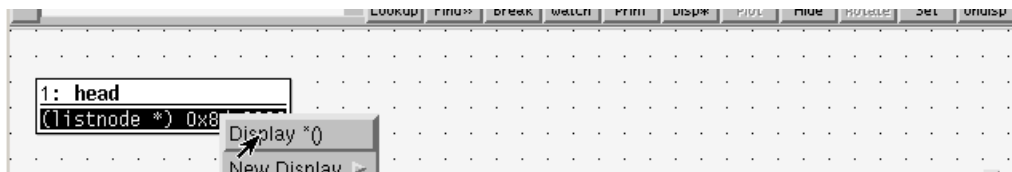△ Breakpoint 1, main () at list2.cc:62
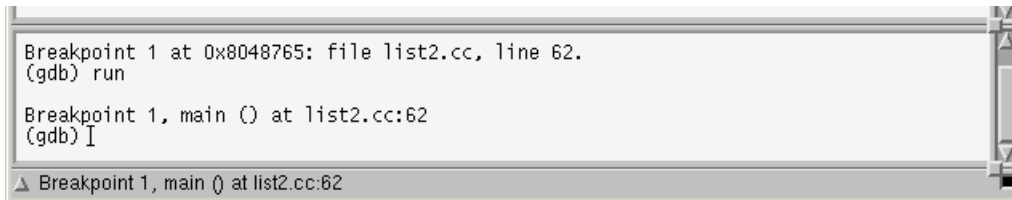
Let's examine the linked list we created. Highlight any occurrence of the variable head in the code, then press and hold the right mouse button to get the popup menu. Select Display head, as shown below.

```
    // Start at head
    p=head:
    whil  Print head
                         id << " ";
          Display head  ;
    }
    cout  Print *head
          Display *head
    ret
}         What is head
          Lookup head
          Break at head
Breakpoi            8765: file list2.cc, line 62.
(gdb) ru Clear at head

Breakpoint 1, main () at list2.cc:62
(gdb)
```
△ Breakpoint 1, main () at list2.cc:62

This causes the "Data Window" to appear and the name and value of head are shown in the window, as shown below.

```
File   Edit   View   Program   Commands   Status   Source   Data                                    Help

0:  head

     Lookup  Find»  Break  Watch  Print  Disp*  Plot  Hide  Rotate  Set  Undisp

    1: head                                                         Data
    (listnode *) 0x8da6008                                          Window

        p->next = NULL;
```

Now we can graphically display the linked list. In the "Data window," press and hold the right mouse button to get the popup menu. Select Display *() to deference the pointer and display the data it is pointing to, as shown below.

```
        Lookup  Find»  Break  Watch  Print  Disp*  Plot  Hide  Rotate  Set  Undisp

    1: head
    (listnode *) 0x8
                  Display *()

                  New Display
```

The "Data Window" now shows the first node in the linked list. The display shows the values of id and next variables in the first node.
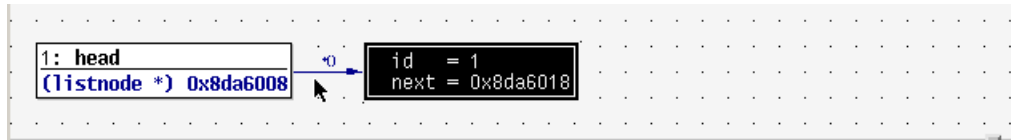
Now highlight the next field of the first node, and using the right button of the mouse, select `Display *()` to deference next and display the second node in the list. It has an `id` of 2 as expected.
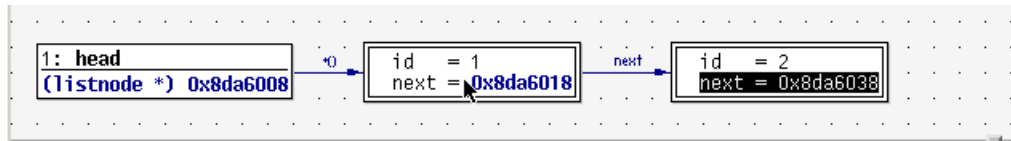
Now repeat this twice to get the third and fourth nodes in the list as shown below.
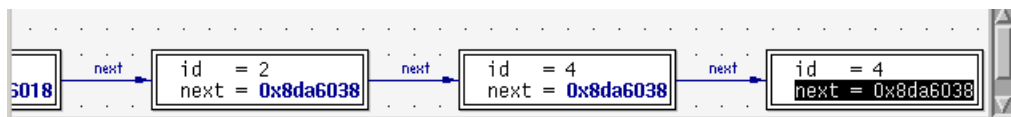
The problem is now clear. The successor to node 2 is node 4 as expected. However, the successor to node 4 is also node 4 (same `id` and `next` values). Thus, the while loop after getting to node 4 goes into an infinite loop.

The cause of the problem must be in how node 4 was inserted on the list. A quick examination shows that the two lines:

```
p2->next = p;
p->next = p2->next;
```

are in the wrong order. They should be:

```
p->next = p2->next;
p2->next = p;
```

In your `xterm` window, edit the program and switch the two lines. Re-compile the program using the same command line as above. Then, in your `DDD` window, click the "Run" button again. Note the `GDB` console and observe that the debugger has recognized that the program has now changed, and runs the new version.
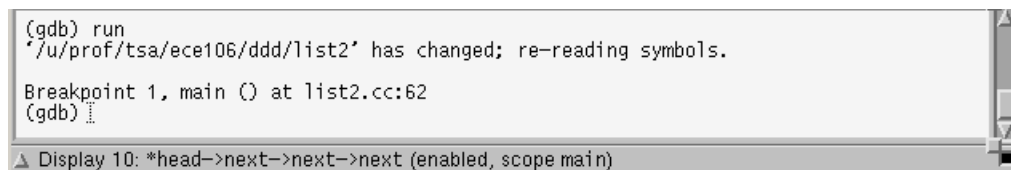
Also, observe how the debugger re-displays the list in the "Data Window" as the program executes. The successor of node 4 is now node 3, as expected, and the program is working correctly. Click the "Cont" button in the "Command Tool," and verify that the program terminates correctly.
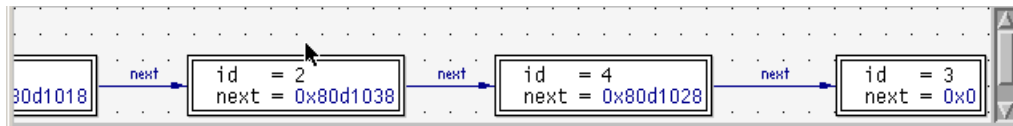
7.4 Part IV – On Your Own

Use a browser to download the C++ program called `harmonic.cc` from the "Labs" section of the course web site. The program prompts the user for an integer n and then computes the nth harmonic number, defined as $1/n + 1/(n-1) + (1/n-2) + \ldots + 1/2 + 1/1$. However, the program prints "`inf`" (i.e., infinity) for any input.

Use `DDD` to debug and fix the program. Even if you think you have spotted the error without the debugger, try to use the debugger to confirm your suspicions.

7.5 Part V – On Your Own

In this part of the assignment, use your browser to download the program `convert.cc` from the "Labs" section of the course's web site. This program converts a temperature from one temperature scale to another. Read through the program to become familiar with its functionality. The program contains a number of compile-time and run-time errors that you must find and correct using the `DDD` debugger. The following shows the "correct" output of the program for a number of inputs, and should serve as a specification for the program.

```
Please enter the temperature you wish to convert,
        followed by its scale (eg. 23 C): 45 F

45 F is equivalent to:
        45 F, 7.22222 C, and 280.222 K
```

```
Please enter the temperature you wish to convert,
        followed by its scale (eg. 23 C): -400 c

-400 C is equivalent to:
        -688 F, -400 C, and -127 K
```

```
Please enter the temperature you wish to convert,
        followed by its scale (eg. 23 C): 100 k

100 k is equivalent to:
        -279.4 F, -173 C, and 100 K
```

Make sure that you do not change the way in which input is read and/or the way in which the output is printed.

Create an executable called `convert` from the `convert.cc` file and test it using the following commands:

```
% g++ convert.cc -o convert
% ~stumm/public/ece244/exercise 1 ./convert
```

Call the exercise multiple times to have it tested with different test cases.

7   Submitting the Programs

Once you are confident that your code is working correctly, it's time to submit it. This is done with the `submitece244f` command. It has the following syntax:

```
submitece244f assignment# filename [filename]
```

You can submit multiple files at the same time simply by listing them on the command line. The command should complete without printing anything. If it does then you have successfully submitted the specified files:

```
% pwd
.../assign1
% submitece244f 1 Hello.cpp convert.cc
```

WARNING: If your submission is incorrect, you will need to resubmit all your files. The command `submitece244f` may output an error message in certain cases. However, `submitece244f` does NOT generate a warning message if you try to submit nonexistent files. Instead, it silently submits empty files under the name of each missing file. Even if you've already submitted some files and then try to resubmit from a wrong directory, the existing submission will be silently overwritten by empty files. This is an extremely dangerous "feature", and for this reason the `submitece244f` command should never be used except immediately following a successful run of `chksubmit`. The use of `chksubmit` is described below.

Another common problem with electronic submission is subtler. If you run the submit command correctly but forget to include a necessary file, you will not receive any error messages. The automarker will not be able to compile and run your program, however, so it will fail every test case.

To prevent these kinds of problems we have provided a program called `chksubmit`. It has the exact same syntax as `submitece244f`:

```
~stumm/public/ece244/chksubmit assignment# filename [filename]
```

However, ***it does not actually submit any files***. Rather, it tries to compile your program given the files specified on the command line. If the compilation fails, or if it does not produce all the expected executables, then `chksubmit` will print an error message. For example, say you forget to include `Hello.cpp` when submitting your files, but rather include `hello.cpp` (lower case "h"):

```
%  ~stumm/public/ece244/chksubmit 1 hello.cpp convert.cc

Attempting to build your code.   Make output follows:

=======================================================
make: *** No rule to make target `Hello.cpp', needed by `Hello'.
Stop.
-------------------------------------------------------
The build appears to have failed.   See the make output for more
information about the details of the error.
-------------------------------------------------------
```

chksubmit tells you that there was a problem compiling your program, and gives you some hints as to where the problem occurred (you will understand this more once you have been fully introduced to the make utility).   If, on the other hand, you include the correct file then the output will look like this:

```
%  ~stumm/public/ece244/chksubmit 1 Hello.cpp convert.cc
Attempting to build your code.   Make output follows:
=======================================================
g++ Hello.cpp -o Hello
-------------------------------------------------------
The build appears to have been successful...

All expected executables were found.   Now run the submit command
as follows:

submitece244f 1 Hello.cpp convert.cc
=======================================================
```

chksubmit tells you that everything worked as expected, and it provides the exact command line that you should use to run the submit command. You should copy and paste that command line to ensure that you run it exactly as shown. Once you have done so you can be confident that you have successfully submitted your code.

8   Deliverables

Submit the Hello.cpp and convert.cc files by using the command:

```
submitece244f 1 Hello.cpp convert.cc
```

Please use the exercise and chksubmit commands to make sure that everything works as expected. Also, you can use the command

```
submitece244f -l 1
```

to be shown a list of the files that you've submitted.