Read the first 6 pages of Chapter 4 of DSO, and part 1 of this handout before you begin this exercise. Then move on and read the rest of Chapters 4 and 5 to prepare for the current project work.

Extra time is given for submission of training work for credit, but it is strongly recommended you do some of this exercise as soon as possible to prepare for Project 2.

Refer to the first training exercises for ddd and other details.

You may structure your program either with a class and member functions or in any simpler way you care to.

1  Begin

In a new, empty directory, begin a C++ program with (1) an integer array A, (2) two size_t variables: LAST and USED, and (3) int variable temp. You may declare and use additional variables. It doesn’t matter what storage extent (static or automatic) they have. Choose A’s length so it is practical for instructive manipulations like those given in textbook examples.

Write a build script.

Make (by reiterating build and editing) your rather trivial program compile and link.

2  Read and Print Half the Array

Code two loops to run successively.

1. Read\(^1\) integers to fill the array approximately half way.

   The code to put each integer into the array should have the form:
   \[
   \begin{align*}
   \text{int } &\text{ temp;} \\
   \text{cin } &\gg\text{ temp;} \\
   \text{A[USED]} &\text{ = temp;} \\
   \text{USED } &\text{ = USED }+\text{ 1;}
   \end{align*}
   \]
   rather than the (better)
   \[
   \begin{align*}
   \text{cin } &\gg\text{ A[USED++];} \\
   \end{align*}
   \]
   because a future exercise might require calling a function to insert the integer. The loop must also update LAST.

   After this loop, the value of LAST must satisfy the invariant:
   // (The value of) LAST is the index of the last array entry filled by the loop, if there is any.

2. Print the integers just read from the array, in the order they were read.

   Build and test your program until it works.

---
\(^1\) All I/O is to be done with standard input and output using the elementary “\$\$” and “\$\$” operators belonging to cin and cout.
3 Prepare for the Experiment with the Array

Run your program under ddd and make it stop just before second loop, where the invariant is true. Details: (1) Start ddd with ddd executableFileName. (2) Set the breakpoint at main. (3) Activate ddd’s “run” operation. (4) Set a new breakpoint in front of the second loop by clicking on a spot in front of that second loop’s first line, and then clicking the stop sign “breakpoint” icon. Observe the new stop sign and message about breakpoint 2 in the (gdb) interaction window. (4) Activate ddd’s “continue” operation, either from the “Program” pull-down menu or by typing continue.

You will of course have to type in the integers.

When your program stops at the second breakpoint, put the array contents on dynamic display: Edit A[0...28] (replace 28 by the last legal index for your array A) into the upper left edit box and then click the Flashlight “display” icon.

You should see the numbers you typed in followed by “garbage” (or zeros if you used a static array.) Scroll and/or enlarge the display window so you can see them.

4 Use the print Command to evaluate expressions

The debugger’s “print” command evaluates and then prints the value of C++ expressions you type in. Try in the (gdb) window:

print 3
print 2003+3
print 5*7
print 5/7
print (double)5/7
print ((double)5)/7
print (int)((double)(5)/7)
print (double)(5/7)

(The last 4 expressions include a C-style cast. For example, (double)5 is the double precision floating point number representation resulting from converting integer 5 to double precision.)

Each value printed is also stored in a debugger history variable of the form $integer. Try:

print $1
print $2
etc...
print $1+$2
print $198

Besides doing calculations, the print command lets you examine the process’s state by printing the values of variables. Try a few now.

5 Printing array element VALUES AND ADDRESSES, Dereference expressions

This material will be new to most of you!! It’s a core topic in Data Structures!
5.1 The "address-of" & operator produces ADDRESS or POINTER values

The C/C++ & operator evaluates to the ADDRESS OF, in other words, the POINTER TO its operand, which must be a variable. Try:

print LAST
See the VALUE of the variable named LAST

print &LAST
(0x... indicates an integer number written in hexadecimal, which is base 16, with the symbols a...f used for digits with values ten...fifteen.) See the ADDRESS of or the POINTER to the variable named LAST.

print &938
See that it is illegal (in fact meaningless!) to request the address of a constant which is not in memory.

A memory address or pointer IS A NUMBER. Examples of such numbers 998, 997, 996, etc. are printed in the figure on page 141 of DSO.

Programmers find it convenient to express numerical address or pointer values in hexadecimal notation. Therefore, gdb (by default) uses the hexadecimal system to display a number when the number comes from an expression denoting an address. But numbers, whether expressed in decimal, roman numerals, tally marks, binary, English (e.g. thirty seven) or hexadecimal are still numbers! Try:

print 938
print /x 938
print 0x3aa
print /x 0x3aa

Take the result of print &LAST and type it after print
Then compare it to the result from print /d &LAST

5.2 Array Elements

The whole array A is a variable, a compound variable. This whole is composed of subvariables which are variables in their own right. Each subvariable is called an array element. So expressions for array elements like A[0], A[1] and A[LEAST] all denote VARIABLES. AS SUCH, the address-of operator on each will produce its own address. Try:

print &A[0]
print /d &A[0]
print A[0]
The last command prints the VALUE of the variable denoted by A[0].

Try it for other elements in the array:

print &A[1]
print /d &A[1]
print &A[2]
print /d &A[2]
print &A[LEAST]
print /d &A[LAST]
print A[1]
print A[2]
print A[LAST]

5.3 Dereferencing

Look back over the contents of the (gdb) prompt window and find some results when print was commanded to print an address. Try to print what is stored at that address by giving the print command on the dereference operator * applied to the address.

Practice this with an address you retype in hex. Also practice this with debugger history variables ($numbers$).

What if you try “print * 0”, trying to dereference the NULL pointer?

Try alternations like print &*something &* &* &*, etc.

5.4 Part 1 Credit

Prepare a credit submission directory for Train2 named Train2

In this directory, create text file containing a copy of the material from the (gdb) interaction window that proves you did the above exercise.

Please eventually “turn in” the your Train2 directory to project Train2 using the command:

```
turnin-csi310 -c csi310 -p Train2 Train2
```

Detailed instructions had been given in Lab1 and in the handout on electronic submission of course work.

---

**Given float * P; what each means**

- **&P**
  - the VALUE of &P
  - is the address of the variable named P (a constant)

- **P**
  - pointer variable P
  - the VALUE of P is what’s stored HERE.
  - It is an address or pointer, because P has type pointer.

- ***P**
  - variable denoted by *P
  - the VALUE of *P is what’s stored HERE.
  - It’s type is float, the type P is declared to point to.
  - (a DIFFERENT variable)

To dereference P means to access the data in HERE.
6 Part 2

You will explore a classic data structure that will be taught in the formal lectures a few weeks from now: The binary tree implemented with linked nodes.

First, copy all the files that are world readable from ITSUNIX cluster directory

"acsi310/Proj2"
to a fresh directory. The files you need to study are printed at the end of the handout.

For this exploration, you will be given a class Mystree interface definition in file Mystree.h and a compiled object file Mystree.o implementation, so the implementation for Mystree’s methods and friend functions will be “hidden”. However, Mystree’s data structures are open.

Our constructor for class Mystree “builds a tree” and stores the address of the root node in public data member Node *PRootNode

(i.e., PRootNode is a pointer to the root node). Each Node is a structure (i.e., a compound variable) with 3 components: One named data holds a single char value, and the other two hold either ADDRESSES OF Nodes or the NULL address value. The NULL address value in C/C++ is numerically equal to zero (0).

Our initial laboratory material codes the creation of a tree holding the string "123456789ABCDEF". Please run our build script (copied into your directory) and run the resulting program which is named main. You will see a crude representation of the 15 node tree holding these characters in “inorder” (a topic for a few weeks from now). The root is printed near the middle of the output, the left subtree with smaller numbers first, about it, and the right subtree underneath. Observe how the numbers (as hexadecimal digits!) increase as you go from top to bottom in the printout.

The laboratory exercise is to modify our main.cpp file so that a smaller, personalized string is used; which will make each student’s work unique, and will make exploring the tree by using DDD to manually DEREFERENCE THE POINTERS a more practical task.

So, first edit main.cpp to replace our 15 character string with your name or a longer or shorter nickname, so the string is between 4 and 6 characters long. Recompile and rebuild the executable program main. Then run main and read of your name from the output.

**For the first credit of Part 2:** Begin running your modified main under the control of DDD. For credit, you will capture a Postscript graphics file containing DDD’s graphical output. Stay tuned!

As usual, first set a breakpoint at main() and command the debugger to “Run”. It will stop just before calling the constructor.

Activate the “NEXT” operation just once.

Now, type M into the input box near the upper left corner of DDD’s main window. M is name of the object of class Mystree which holds a pointer to the root of the tree the constructor had built.

Some command buttons, including “Display”, to the right of this box take an operand from this box. Click the “Display” button, which is labeled with a flashlight icon. You should see a new pane appear at the top of DDD’s window. Now try to make that pane bigger by dragging the “resize” handle which is a small square near the right lower corner (this direction:→) of the new pane.

You should see a hexadecimal numeral in one box within the new “display” pane. The corresponding number is the value of the one data member of instance P; that data member is a pointer to the root node of the tree.

Double (left) click on this pointer value. This commands DDD to DEREFERENCE the pointer and display THE OBJECT IT POINTS TO. (Remember that an object is a variable.) You should quickly notice that this object, the variable the pointer refers to, has 3 components.

One of the 3 components is a variable of char type. Its value is displayed in two ways: (1) the numerical value, and (2) the letter, number or other printable character image or “glyph” if the
numberical value codes a printable character according to the ASCII standard.

The other 2 components are pointers, i.e., pointer type variables. NOW: Double click first on one pointer, and then on the other.

You should see two more 3 component displays. The class member names PLeftNode and PRightNode are displayed alongside ARROW LINES drawn from the root node to the two new displays.

Use the right button to drag the “Left node” somewhere to the left and then the “Right node” somewhere to the right.

To complete the exercise, move displays of Node around to make a nice left-to-right tree layout; and dereference all non-NULL pointers and continue laying out the tree until the entire tree is on the display. The figure 10.5 in DSO will give you the idea of what kind of display to make.

Verify that when you read the chars stored in Nodes, left-to-right beginning at the bottom leftmost node, you can read off the name or nick-name string you coded in main.cxx.

When this is confirmed, “snap a picture” to make a Postscript file which we can look at to give you credit. For the first credit, name your picture part2.ps and put it in the directory Train2.

Follow the directions below to create the file named part2.ps

7 Snap a Picture!

In future lab or debugging work, you might have changed some displayed data structures before you decided to print or save the graphical display. In such cases, do a last refresh.

Make a final rearrangement so what you want to present is clear. Resize the display pane if necessary. Finally, click anywhere outside a displayed box to make sure none of the boxes remains “selected” (otherwise, your picture will be only the selected box, not the entire display.)

1. Pull down the the File menu, activate “Print graph...” (or just type Ctrl-P if you prefer), and observe the DDD:Print dialog window.

2. Select “File” in this dialog.


4. Optional: Edit the name dddgraph.ps to make it what you want; this is necessary if you do not want to overwrite (and lose) pictures you snapped earlier.

5. Check/fix the contents of the dialog box and when they are OK, click “Print” to save a file copy of the display image.

6. Verify the file with the name you did choose and “extension” .ps now appears in the working directory of your ddd run. (Use Unix command ls of course.)

7. You can preview the image with Ghostview, installed on itsunix, with:
ghostview list1.ps for example, if your “picture file” was named list1.ps. Give this command (like ls) from the shell.

8. If you are in the lab, you can print it with the non-ddd command “lpr -Pcolorlab list1.ps”
You can also use any other Unix printing methods you know and perhaps have paid for.
8 Another Train 2 Credit: Explore the tree with text-based GDB commands.

Go back to the beginning and review how to use typewritten commands to observe the values of variables and to dereference pointers.

Use them to fully explore the tree; the numeric address of the root node can be viewed with the command `print M.PRootNode` and the root node with its component values can be viewed with `print * M.PRootNode`.

Now observe the hexadecimal display of the value of M.PRootNode. Let’s call it 0xXYZZZYZ for illustration’s sake. Try observing the SAME NODE you had viewed viewed with “print * M.PRootNode” by the following command:

```plaintext
print *(Node *) 0xXYZZZYZ
```

What are you doing here? The cast operation (Node *) tells the interpreter in gdb to interpret the number 0xXYZZZYZ as the address of a Node type variable. The subsequent print * tells gdb to print the value of the Node type variable with that address.

Try and compare: `print (unsigned int) 0xFFFFFFFF` with `print (int) 0xFFFFFFFF`. To finish, give enough (gdb) commands to fully explore the tree, so that you find every name/nickname character (copy) within it².

You may find it efficient to use gdb history variables (discussed under part 1, they begin with $) to hold pointer values before you give the command to dereference them. For example,

```plaintext
(gdb) print M
$7 = {PRootNode = 0x80acbc0}
(gdb) print $7.PRootNode
$8 = (Node *) 0x80acbc0
(gdb) print * $8
$9 = {data = 56 '8', PLeftNode = 0x80acbd0, PRightNode = 0x80acc40}
(gdb) print * $9.PLeftNode
$12 = {data = 52 '4', PLeftNode = 0x80acbe0, PRightNode = 0x80acc10}
```

When you are done, make a text file named part2.txt that contains text copied and pasted from the (gdb) interaction window and put it in directory Train2.

9 A Helpful Tip

Many of you will find the usage of * (for dereferencing), . (“period”, for component selection), -> (for their combination), & (for address-of), etc., in C/C++ a bit difficult. The debugger text interaction facility is very helpful for playing with combinations of these operations, to see what they do and when their use is erroneous. I found that so when I first had to teach this stuff!

10 Guess What—you’re not done!

The instructions given in great detail are intended for first-time DDD users. It is expected that in the future, you will be able to perform these operations, without the already given instructions repeated, within future labs and projects.

²I wrote copy here because a string might contain REPEATED LETTERS!
So, it would be wise to practice this and similar debugger skills until you can repeat all the exercises in this handout with little or no reference to the instructions. (At least until the end of the semester, and so you can quickly relearn for CSI333, CSI402, etc.) It might be helpful to write in a notebook (or in the space below) key tips to remember to use in the future.
11  Train2 Part 2 File Listings

11.1  main.cxx
#include <iostream>
#include "Mystree.h"
using namespace std;
main()
{
    Mystree M("123456789ABCDEF");  // Note: The string's length 15 == 2^4 - 1
    cout << M;  // Print a nice, balanced tree..
    return 0;
}

11.2  Node.h
#ifndef Node_h_included
#define Node_h_included
class Node {
public:
    char data;
    Node *PLeftNode;
    Node *PRightNode;
};
#endif

11.3  Mystree.h
#ifndef Mystree_h_included
#define Mystree_h_included
#include <iostream>
#include "Node.h"

class Mystree {
public:
    Mystree(char *PCstring = NULL);
    // Pre: PCstring == NULL or it points to a C-string (null terminated
    //      array of chars)
    // Post: If PCstring != NULL, PRootNode points to the root
    //      of a binary tree of non-null chars and the in-order traversal of this
    //      tree equals the C-string pointed to by PCstring.
    //      If PCstring == NULL or the C-string is "" then PRootNode == NULL
    Node * PRootNode;
    friend std::ostream& operator<<(std::ostream& outs, Mystree& target);
    // Post: If target.PRootNode == NULL nothing is appended to outs.
    //      Otherwise a cool (ugh?) display of the chars in the
    //      binary tree rooted at *(target.PRootNode) is printed in
    //      reverse-inorder.
};
std::ostream& operator<<(std::ostream& outs, Mystree& target);
#endif