```
======================================================================
  CS 50011           INTRODUCTION TO SYSTEMS II          Summer 2017
                               Lab #2
======================================================================
```

Name: _____________________________________

Part 0: Introduction

1. Create a Lab02 directory inside your home directory. All solutions to the programming exercises should be stored here.

2. Copy this week’s lab files (all files in ~jeff/labfiles/lab02) into your Lab02 subdirectory.

3. Complete the tutorial up through the “Exploit” section located at https://turkeyland.net/projects/overflow/index.php

==
Part 1: Shell Code (25 points)

[20 points] Come up with two alternative pieces of shell code that accomplish the same task as the chmod() shell code in the tutorial. Place each in “myshellcode1.c” and “myshellcode2.c”. It should be in the following format:

```
int main() {
__asm__(
“your_assembly_here”
);
}
```

[10 points] Write the hexadecimal payload for each in the file “payload”.

==

Part 2: Hack a grading program (25 points)

Programs are not necessarily written in a secure manner – allowing individuals to make them do things that are entirely unexpected.

You are provided with a sample “grading module.” Below is what the test() function looks like:

```
static int test()
{
  char Garbage[*];
  int i;
  int Score = 75;
  YourFunc();
  i = 7;
  Score = 50;
  return Score;
}
```

* The size of the character array Garbage is determined by your login and often times different from that of other students.

Hack the module for 100.

Note that after you link against the grading module, you should run the executable in the following manner:

$ ./executable_name yourlogininlowercase

This is how your program will be graded.

Functions to write:

void YourFunc();

Do whatever you want to make the test module give you a score of 100.

hack_test.o is included in the labfiles directory. You should link against it.

==
Part 3: Advanced C (50 points)

The purpose of this part is to give you experience in using generic functions to do traversal of trees. You'll customize the functions by passing in pointers to functions that will do comparisons and operations. We'll also be using storage class qualifiers to indicate which data are modifiable or not.

You know how to write functions to build, maintain and traverse trees but, if you're going to ever be able to build large software projects, it would be convenient to be able to encapsulate these functions in a library and just use them instead of having to write them over and over again. When you're writing a program, you'd like to be able to say "I'd like a tree of gadgets there." Some programming languages have mechanisms to permit the programmer to write something once and then use it for many things thereafter. If you've ever seen or heard of C++ **template classes** you'll know what we're talking about.

Even with languages that support **reuse** of data structures and algorithms, the programmer must take care to make sure that the resource to be reused will behave as expected with no bizarre side effects. That often turns out to be a very tedious undertaking. Take a look at the code for the C++ STL (Standard Template Library) sometime if you want to see a good example.

Since this class involves less with programming than it does with simply understanding how the computer works, we're not going to just give you code to do generic tree traversal and manipulation and have you write programs using it. We're going to ask you to write the generic functions.

Actually the things you write here will not be completely generic. We'll only work with **one** kind of tree node. To do more than this, we'd have to use casts. Once you say "goodbye" to strongly typed programs, you also say "goodbye" to ease of debugging.

The data structures are declared this way:

```
typedef struct Person
{
  char         *Name;
  unsigned int  Phone;
  char         *Office;
  char         *Login;
  char         *Title;
} Person;

typedef struct Tree
{
  struct Tree   *Left_Ptr;
  struct Tree   *Right_Ptr;
  struct Person *Data;
} Tree;
```

```
typedef enum {
  LNR,
  RNL,
  NLR,
  LRN
} Tree_Traverse_Method;
```

Examine "lab02.h" carefully. It contains declarations for the data structures you'll use as well as prototypes for the functions you'll write.

There are three basic operations that you might want to do with trees: Insertion, Traversal and Deletion. Deletion is pretty straightforward but Insertion and Traversal are ones you might want to customize. For instance, if the data you're inserting has multiple fields, which one do you want to sort by? If you're traversing a tree, what do you want to do with it? It would be convenient to write functions that generically insert and traverse.

You should write the following functions:

```
Person *Person_Create (const char *, unsigned int, const char *,
                       const char *, const char *);
```

This tedious function will dynamically allocate memory for a **Person** as well as space for each of its pointer fields. Do the obvious things to make sure that each of the arguments is valid and that each allocation was a success. Return the pointer to the newly allocated structure.

```
void Person_Delete(Person **);
```

This function accepts a pointer to a pointer to a **Person** structure that is to be freed. The function should also free every field in the **Person** structure. It should also set the pointer that the argument points to to NULL.

This function should assert that the argument is not NULL.

```
Tree *Tree_Create (Person *);
```

Dynamically allocate memory for a **Tree** structure. Copy the argument into the **Data** field. The **Left_Ptr** and **Right_Ptr** fields should be initialized to NULL. The function should return a pointer to the newly allocated element. Note that the only memory allocation to be performed for this function is the one to allocate the **Tree** structure.

This function should assert that the pointer argument is not NULL.
This function should use assert that the memory allocation is not NULL.

```
void Tree_Insert (Tree **, Person *,
                  int (*)(const Person *, const Person *));
```

The first argument is a pointer to the pointer to the root **Tree** element. The second argument is a pointer to the new **Person** structure to be inserted. The third argument is a pointer to a function that will be used to order the insertion. Basically, this function will usually be called this way:

```
Tree_Insert(&Root, Person_Create(...), Sort_Person_By_Name);
```

This means that you can call this function initially when **Root** is NULL and it should set the **Root** variable to point to the newly inserted element. When the proper place to insert the new **Person** structure is found, create a **Tree** structure, to hold the pointer to the **Person** structure and attach it to the tree.

You should assert that none of the arguments are NULL.

```
Tree *Tree_Find(Tree *, const Person *,
                int (*)(const Person *, const Person *));
```

This is a recursive function that will search for a **Tree** node whose **Name** field matches the second argument. If a matching node is found, the function should return a pointer to the node. If no match is found, the function should return NULL. This function will most likely be invoked in the following way:

```
Person_Ptr = Tree_Find(Root, &One_Person, Sort_Person_By_Name);
```

The function should assert that the second and third arguments is not NULL. It's OK for the first argument to be NULL.

```
void Tree_Traverse(Tree *, Tree_Traverse_Method, void (*)(Person *));
```

This function accepts a pointer to the root element of a tree to be traversed. The second argument is an indicator that says in what order the tree is to be traversed (e.g. LNR: left-node-right, RNL: right-node-left, LRN: left-right-node or NLR: node-left-right). The third argument is a pointer to a function that says what to do with a **Person** field. For instance, one might create a function that printed the **Name** field and then pass it as the third argument.

```
void Tree_Delete(Tree **);
```

Delete the entire **Tree** and its **Person** fields recursively. The function should call **Person_Delete** to delete the **Person** fields. After the function returns, the root of the tree that was passed by pointer should be set to NULL. e.g. if the function is called this way:

```
Tree_Delete( &Root );
```

then **Root** should be set to NULL after the function is finished. The function should assert that the first argument is not NULL. However, keep in mind that it's OK for the first argument to point to a NULL pointer.

int Sort_Person_By_XXXX(const Person *, const Person *);

You'll create a function like this for each field in the **Person** structure where "XXXX" is replaced by the name of the field. These functions should work like strcmp():

- If the field of the element in the first argument is less than the same field of the second argument, return a negative number.
- If the field of the element in the first argument is equal to the same field of the second argument, return a zero.
- If the field of the element in the first argument is greater than the same field of the second argument, return a positive number.

This function should assert that neither argument is NULL.

We’ve supplied an input file called "Faculty" that you can use for lab02_main.c.

We also provide a header file, "lab02.h" for you. It contains prototypes for each of the functions that you will write as well as #definitions.

Note that since you're not using any file I/O operations, you don't need to include stdio.h. For this assignment, the only other header files you'll need are malloc.h, assert.h and string.h.

There are no error codes for this assignment but watch out for the **Tree_Traversal_Method.**

To start working, make sure you have copied the Makefile from ~jeff/labfiles/lab02 into your account and write lab02.c. When you are ready to try your work, type either “make lab02_main” or “make lab02_test” to build programs that can be used to test your code. Note that we will not provide the lab02_test.o file for you immediately. You should first practice using lab02_main.c to test your code.

Standard rules:

- You may add any #includes you need to the top of your hw11.c file.
- You may not create any global variables other than the ones that are provided for you. Creation of additional global variables will impact your grade.
- You should check for any failures and return an appropriate value.
- You should not assume any maximum size for the input files. The test program will generate files of arbitrary size.
- Don't look at anyone else's source code. Do not work with any other students.

The operation of your functions will be graded out of 100 points. The point breakdown will be determined by the test program.