CSCE 155 - C

Lab 06 - Functions, Passing By Reference, and Enumerated Types

Dr. Chris Bourke

Prior to Lab

Before attending this lab:

1. Read and familiarize yourself with this handout.
2. Review the lecture notes on conditionals, or the following free textbook resources:
   - Functions: `http://www.cs.cf.ac.uk/Dave/C/node8.html`
   - Enumerated Types: `http://www.cs.cf.ac.uk/Dave/C/node9.html`
   - Pointers: `http://www.cs.cf.ac.uk/Dave/C/node10.html`

Peer Programming Pair-Up

To encourage collaboration and a team environment, labs will be structured in a *pair programming* setup. At the start of each lab, you will be randomly paired up with another student (conflicts such as absences will be dealt with by the lab instructor). One of you will be designated the *driver* and the other the *navigator*.

The navigator will be responsible for reading the instructions and telling the driver what to do next. The driver will be in charge of the keyboard and workstation. Both driver and navigator are responsible for suggesting fixes and solutions together. Neither the navigator nor the driver is "in charge." Beyond your immediate pairing, you are encouraged to help and interact and with other pairs in the lab.

Each week you should alternate: if you were a driver last week, be a navigator next, etc. Resolve any issues (you were both drivers last week) within your pair. Ask the lab instructor to resolve issues only when you cannot come to a consensus.

Because of the peer programming setup of labs, it is absolutely essential that you complete any pre-lab activities and familiarize yourself with the handouts prior to coming to lab. Failure to do so will negatively impact your ability to collaborate and work with others which may mean that you will not be able to complete the lab.

1 Lab Objectives & Topics

At the end of this lab you should be familiar with the following

- Basics of enumerated types
- Functions with pass-by-value and pass-by-reference parameters
- How to design and use functions with error handling
- How to modularize code into separate header and source files
- The proper use of enumerated types and functions in solving a given problem

2 Background

Enumerated Types

Enumerated types are data types that define a set of named values. Enumerated types are often ordered and internally associated with integers (starting with 0 by default and incremented by one in the order of the list). The purpose of enumerated types is to organize certain types and enforce specific values. Without enumerated types, integer values must be used and the convention of assigning certain integer values to certain types in a logical collection must be remembered or documentation referred to as needed. Enumerated types provide a human-readable "tag" to these types of elements, relieving the programmer of continually having to refer to the convention and avoiding errors.

Passing by Value versus Passing by Reference

In general there are two ways in which arguments can be passed to a function: by value or by reference. When an argument (a variable) is passed by value to a function, a copy of the value stored in the variable is placed on the program stack and the function uses this copy's value during its execution. Any changes to the copy are not realized in the calling function but are local to the function.

In contrast, when an argument (variable) is passed by reference to a function, the variable's memory address is used. Locally, the variable is treated as a pointer; to get or set its value, it must be dereferenced using the `*` operator. Any changes to the memory cell pointed to by that pointer are reflected globally in the program and more specifically, in the calling function.

A full example:

```c
//a regular variable:
int a = 10;

//a pointer to an integer, initially pointing to NULL
int *ptr = NULL;

// & is the referencing operator, it gets the memory
// address of the variable a so that ptr can point to it
ptr = &a;

// * is the dereferencing variable: it changes the pointer
// back into a regular variable so a value can be assigned
// or accessed:
*ptr = 20; //a now holds the value 20
```

One of the main uses of pointers is to enable pass-by-reference for functions. Consider the following function:

```c
1 int foo(int a, int b) {
2   a = 10;
3   return a + b;
4 }
```

When we call `foo`, the variable's we pass to it are passed *by value* meaning that the value stored in the variables at the time that we call the function are copied and given to the function. The change to the variable `a` in line 2 *only affects the local variable.* The original variable is unaltered. That is, the following code:

```c
int x = 5;
int y = 20;
int z = foo(x, y);
printf("x, y, z = %d, %d, %d\n" x, y, z);
```

would print `5, 20, 30`: the variable `x` is not changed by the function `foo`. Contrast this with the following:

```c
1  int bar(int *a, int b) {
2    *a = 10;
3    return (*a) + b;
4  }
```

In the function `bar`, the variable `a` is passed *by reference* using a pointer. Since the memory address is passed to `bar` instead of a copy of a value, we can alter the original value. Now when line 2 changes the value of the pointer `a`, the change affects the original variable. That is, the code

```c
int x = 5;
int y = 20;
int z = foo(&x, y);
printf("x, y, z = %d, %d, %d\n" x, y, z);
```

Will now print `10, 20, 30` since the variable `x` was passed by reference.

Passing by reference means that we can "return" multiple values and/or use the return value to indicate an error code instead of a value now.

3 Activities

Clone the GitHub project for this lab using the following URL: https://github.com/cbourke/CSCE155-C-Lab06

A flower shop sells various arrangements of a dozen flowers (roses, lilies, daisies) in two colors each (red or white) with a choice of bouquet or vase. You are given an (incomplete) source file, `florist.c`, of a program that takes the order for a flower arrangement from the user and displays the cost. The program uses three enumerated types to define the type of flowers, color and flower arrangement. Your task is to complete the program and implement the getCost function to compute the cost based on the following rules.

1. The base price for each of the flowers is described in Table 1

Flower	Price
Roses	\$30
Lilies	\$20
Daisies	\$45

Table 1: Flower Prices

2. Red Lilies and Red Daisies have an added cost of \$5 and White Roses have an added cost of \$10. There is no additional cost for other flower/color combinations

3. Bouquets are free of charge while vases add an additional $10 charge to the total

3.1 Defining Enumerated Types

To complete the program, perform the following tasks.

1. Insert the required items in the 3 enumerated types. The `Color` enumerated type has already been completed for you. Note: the rest of the program assumes that all enumerated types begin with 1, you should keep this assumption so that you don?t need to change the block of `printf` and `scanf` menu statements.

2. After taking input from the user, the program calls the function `getCost` to determine the cost. This function takes three input parameters, the flower type, the color and the arrangement and returns the cost (a `double`).

   a) The function prototype has been completed for you. Implement the definition of this function to return the cost of a given arrangement using the costs shown on the price list.

   b) In the main function, declare the other required variable(s), complete the function call with appropriate arguments and print the cost. What's the data type of the variable cost?

3. Compile using `gcc -o florist florist.c` and test your program. Answer the questions in your handout.

3.2 Passing By Reference

Due to a harsh winter, Red Daisies are no longer available. We will make the appropriate changes to the `getCost` function to accommodate this change and to communicate errors through the function's return type. The cost will be communicated to the calling function by reference rather than as its return value.

1. Change the signature of the getCost function to:
   `int getCost(double *cost, Flower flower, Color  color, Arrangement arr)`

2. Change the definition of the function to do the following:

   a) If given invalid arguments, it returns a value of 1 (indicating an error)

   b) If given valid arguments, it returns a value of 0 (indicating no error) and places the cost in the cost variable (which is passed by reference)

3. In the `main` function, make other necessary changes to accommodate the above. In addition, you should check the return value of the `getCost` function and print an error message in the case that it results in an error.

4. Test your changes and demonstrate them to a lab instructor.

3.3 Using Error Codes

Enumerated types are useful, but safety is not always guaranteed. To illustrate, the following code would compile: `Color c = 10;` even though there is no color corresponding to the integer 10. What happens if you call your `getCost` function using such a color? To prevent this, we will add additional code to detect and handle this type of error situation.

1. Add another enumerated type called `Error` that will recognize the following error types: No Error, Unavailable Selection Error, Invalid Argument Error, and Null Pointer Error (note: a good convention is to use all-caps with words separated by an underscore).
2. Add additional code and checks to your `getCost` function to check for valid input (on all the arguments).
3. Modify the `getCost` method to return one of your enumerated types based on the type of error encountered.
4. Modify your `main` function to check for the type of error returned and print an appropriate error message
5. Demonstrate your working program to a lab instructor

4 Modularizing Your Code

Make your code more modular by organizing components into separate files.

1. Place all of your function prototypes and enum definitions in a header file, `flowers.h`
2. Place all of your function definitions into a source file, `flowers.c`
3. Add the appropriate preprocessor directive to `flowers.c` and `florist.c`
4. Compile and link your files:
   a) Compile (but do not link) the `flowers.c` source file into an object file using: `gcc -c -o flowers.o flowers.c`
   b) Compile your `florist.c` source and link into the `flowers.o` object file using: `gcc -o florist flowers.o florist.c`
5. Demonstrate your results to a lab instructor.

5 Advanced Activity 1 (Optional)

In the C language, there is no built-in or standard mechanism to print a human-readable value of an enumerated type. For example, given an enumeration:

```c
enum {
  RED = 1,
  WHITE
} Color;
```

there is no built-in function to print the enumeration constant “red” or “white” given a `Color` type.

Modify your program by adding a function or functions so that you are able to print he labels of the enumerated items. As a hint: the function signature for `Color` should be something like `char * getName(Color c);`

6 Advanced Activity 2 (Optional)

Large projects require even more abstraction and tools to manage source code and specify how it gets built. One tool to facilitate this is a program called Make which uses makefiles–specifications for which pieces of code depend on other pieces and how all pieces get built and combined to build an executable program (or collection of executables). The use of modern IDEs has made the build process much more manageable, but make is still widely used. Several useful tutorials exist describing how makefiles can be defined and used.

- http://www.opussoftware.com/tutorial/TutMakefile.htm
- http://mrbook.org/tutorials/make/

Below is an example makefile. Copy and paste this text into a file called `makefile` in the same directory as your other code. From the command line, type `make` and observe that all of the elements of your program are built automatically!

```makefile
#Example makefile
#define a variable, CC which defines the compiler and its flags
#so that we can reuse it
CC = gcc -g -Wall

florist: florist.c flowers.o
	$(CC) -o florist flowers.o florist.c

```

```
flowers.o: flowers.h flowers.c
  $(CC) -c -o flowers.o flowers.c

clean:
  rm -f *.o *~ florist
```