Computer Science 1MC3
Lab 6 – Procedures, Functions and User Defined Data Types

It is strongly recommended that the lecture summaries on functions and procedures be read before attempting the lab assignment questions.

Procedures

Procedures are used in C to make it easier to repeat certain tasks or code. For example you may want to have a procedure to print out some employee information to the screen:

```
void print_header (int emp_num, int start_date) {

      printf("Emp. number: %d \n Start Date: %d", emp_num, start_date);

}
```

Which would print out say:

```
Emp. Number: 9999
Start Date: 2003
```

This procedure may be used many times, printing out whatever data you want. Now, you may be wondering how you would actually make the procedure happen. This is what we mean when we say were "calling" a procedure. You would do this from your main program in this fashion

```
print_header(9999, 2003);

~or~

int employee=9999, start=2003;
printheader(employee, start);
```

It is also worth noting that procedures can call other procedures.

Functions

Functions are basically procedures, but with one main difference. Functions, unlike procedures, must return a value before the function terminates. A function in computing can be thought of like a function in math. You give a function a value, which it then manipulates, and returns.

The syntax for a function is as follows:

```
Data_type fuction_name (data_type1 input1,. . .,data_typeN inputN) {

      Code to be executed;

      return variable of Data_type;

}
```

So say you needed a function to take the factorial of something. Recall: `4!=4*3*2*1`. It may be implemented in the following fashion.

```
int factorial (int x) {

      int index;

      for(index=1; index<x; index++){
            x=x*(x-index);
      };

      return x;

}
```

You could call this from your main program like:

```
int x;

x=factorial(4);
```

Please note that a function can return any data_type you want, and may accept as many values as you want.

A last note about procedures and function

It is necessary to declare your functions and procedures before your main program so the computer can reserve memory. For example say the above two function were to be used in a program the program would look like this:

```
 void print_header (int emp_num, int start_date);
int factorial (int x);

int main (void) {

      code
      retrun 0;

}
```

User Defined Types

In c it would seem that you are constricted to the data types: integer, float, and character. However, this isn't the case. As programmers, you have the option of declaring your own data types. For instance if you wanted a variable to represent a day of the week it may be useful to declare a type which contained all the days.

```
enum week {sun, mon, tue, wed, thu,  fri, sat};
```

Now to declare a variable of this new type, you must use something called `type_def` which is used to define a new type.

```
typedef enum week day_type;
day_type day=mon;
```

Now you have a variable day that may be treated like any other variable. The same way you wouldn't store anything else but integers in an `int x` you may only store the predefined values of `sun. . .sat` in your `day_tpe`. Here is a sample program which uses our `day_type`.

```
enum week {sun, mon, tue, wed,  thu, fri, sat};
typedef enum week day_type;

int main (void) {

      day_type day=mon;

      if (day==mon) {
            go_to_doctor();
      } else (day_type++);

}
```

Saying `day++` simply tells the computer to go to the next value, in order of the way we declared, of the `day_type`. So if `day=mon`, `day++` would be equal to `tue`.

Structures

The structure mechanism provides you a means of combining different types of variables into one. For instance, you may want an employee structure to hold all the information about on given employee.

```
struct emp {
      int emp_num;
      int salaray;
      int start_date;
};
```

Then we may declare variables using this structure using `typedef`.

```
typedef struct emp emp_type;

emp_type employee1, employee2, employee3;
```

Would create 3 employees of the employee structure.

Accessing Members of A Structure

Using the previous section's declarations we may initialize the employees by:

```
employee1.emp_num = 1000;
employee1.salary = 45000;
employee1.start_date = 2001;
```

We can thereby ask for any value by saying `var_name.stuct_part;` For instance:

```
if (employee1.salary<50000) {
      Give_raise();
}
```

A Final Note on struct and types

It is possible to use user defined data types in your structures. For instance:

```
enum game {rock, paper, scissors};
enum result {win, loss, tie};

typedef enum game game_type;
typedef enum result result_type;

struct playing {
      game_type choose;
      result_type outcome;
}

typedef struct playing playing_type;
```

And furthermore you can declare an array of any declared type!

```
playing_type simulation[10];
```

In which then you could say:

```
simulation[1].choose=rock;
simulation[1].outcome=loss;
```

Homework

1. Design a function to take x and put it to the exponent y. Implement this in a program (test the function by making a program and running it).

2. Make an enumerated date type for all the months in a year. Declare a variable "`testing`" of this data_type and play around with it. What happens when:

You try to put an integer value into testing?

You say `testing=testing+3;`

3. Make a structure and declare two variables of this structure (a and b). When will `a==b` be true?