Patrick Harrington
ECEN 2020
Homework 1
September 8, 2013

1 Variable Types

(a) Size of integer on computer and Arduino

The size of integers on my computer (in bytes) is 4 bytes (found by using C's sizeof(int) function). On the Arduino, the integer size is smaller at 2 bytes.

(b) What is a type cast? Why is it useful?

A type cast is an explicit re-definition of a data value's type. Type casting would allow, for instance, an integer to be converted into a double. The data itself remains unchanged, but the format the data uses changes. This has many uses in arithmetic operations. One example is truncating doubles or floats, or for more clever bit-bashing.

(c) Simply explain what preprocessor directives are and what they can be used for

A preprocessor directive is used most often to include files or to set up global variables. In C, preprocessor directives are initiated with a "`#`" symbol before the `main` function.

2 Explain the Following Section of Code

```
1 int* x;
2 int* y;
3 x = malloc(sizeof(int));
4 y = malloc(sizeof(int));
5 *x = 1;
6 *y = 2;
7 x=y;
8 printf(``%d, %d\n'', x ,y);
```

First, x and y are initialized as pointers to `ints`. In lines 3 and 4, x and y are reserving data that is exactly an integer's size in bytes. Running this code on my laptop would allocate 4 bytes off the heap. In lines 5 and 6, the pointers `x` and `y` are each assigned values in memory. In line 7, point `x` is then set to point to `y`. When the program is ran, two numbers are printed because of line 8's `printf` call. The numbers are equal, and represent the value that `y` points two with a memory address at 2. This can be seen from the output of `printf(‘‘%d, %d n’’, *x, *y);` if ran before and after the assignment at line 7.

3 Loops

```
1 void setup() {}
2 void loop() { my_function();}
3 void my_function()
4 {
5   static int MyVar1 = 0;
6   int MyVar2 = 0;
7   MyVar1++;
8   MyVar2++;
9 }
```

(a) What are the values of `MyVar1` and `MyVar2` at the end of `my_function` for the first five calls to `my_function`?

At the end of five calls to `my_function`, the variable `MyVar1`=5, and `MyVar2`=0. If both variables are set to be `static`, they increment at the same time. This is to be expected, as the variable in line 6 is re-initialized every time the loop is ran through.

(b) What would the values be if these variables were the same type, but global?

If both variables are global, that is, if they are set before `void setup()`, the output is quite different. Both `MyVar1` and `MyVar2` increment up at the same time. At the end, both variales are equal to 5.

4 Given the Following Variables

```
const boolean bPenguin  = true;
const boolean bFrog     = false;
const int iTurtle       = 0x19;
const int iRabbit       = B00001111;
const int iHampster     = 0;
```

(a) What would the result of the following statements be?

```
iTurtle   &   iRabbit;
iTurtle   &&  iRabbit;
bPenguin  &&  iRabbit;
iHampster &   iTurtle;
iTurtle   ||  iRabbit;
iTurtle   |   iRabbit;
iTurtle   |   bFrog;
iHampster ||  bPenguin;
```

The output of each line is as follows:

1. 9
2. 1
3. 1
4. 0
5. 1
6. 39
7. 25
8. 1

5 Explain the Result of the Following Code

```
union data {
  unsigned char temp;
  unsigned int  time;
};
int main(){
  union data myData;
  myData.time = 0xFCAB;
  myData.temp = 0x0;

  printf(``Address of myData.time: '');
  printf(``%x\n'', (int)&myData.time);
  printf(``Address of myData.temp: '');
  printf(``%x\n'', (int)&myData.temp);
  printf(``\n'');
  printf(``Value of myData.time: '');
  printf(``%x\n'', myData.time);
  printf(``Value of myData.temp: '');
  printf(``%x\n'', myData.temp);
  return 0;
}
```

The `union` data type is a bit exotic; this data type stores all data in the same piece of memory. In this case, both variables in the union are unsigned `char` for temp and `int` for time, respectively. The output of this code returns the same address for both variables in the union, which is consistent with the properties of the `union` data structure.

The values, however, are a bit peculiar. When formatted to output in hexadecimal values, `myData.time` is `FC00` and `myData.temp` is `0`. This is because the last two bytes are overwritten by `myData.temp`.

6 Adding Integers in Two Different Ways

(a) Pass by Reference

In this code, two integers are added by referencing. The first argument is set to be the sum of the first argument and the second argument.

```
int Add_PassByRef(int *x, int *y){
  *x += *y;
}
```

(b) Pass by Value

Here, the values are placed on the stack, added and returned. Unlike in the previous sum function, this one leaves both arguments unchanged.

```
int Add_PassByValue(int x, int y){
  return x+y;
}
```