Part B: Terminology – worth 1 mark each

FILL IN THE BEST ANSWER—INSERT ONE COMPLETE WORD IN THE EACH SPACE PROVIDED BELOW. NOTE THAT YOU MUST USE THE CORRECT WORD IN EACH SPACE; NO 'APPROXIMATIONS'

28. Just prior to the linking phase of compilation, the compiler outputs an _____*object*_____ file; several such files are combined together by the linker to form an executable file.

29. Each makefile consists largely of a set of ____*rules*_____, which in turn have two components: a line listing dependencies and a/n ___*command* or *action* _____line.

30. `system()`,`pause()` and `sleep()` are all examples of Linux-based ____*system*____ calls that can be called from within you C programs.

31. The '*****' symbol used in pointers is referred to as the _____*dereferencing*_____ operator.

32. The power of pointers comes from the fact that they allow the programmer to access a CPU's

_____*indirect addressing*_____ mode.

Part C: ANSWER THE FOLLOWING QUESTIONS IN THE BOXES INDICATED. IN MOST CASES, THE SPACE PROVIDED SHOULD BE MORE THAN ADEQUATE TO HOLD YOUR ANSWER

33. Pointers and arrays operate almost identically in C, but there are differences. In the box below list three ways in which pointers differ from arrays (aside from obvious notational differences.)

1. A pointer is always 4 bytes (on a 32-bit architecture); an array is sized as many bytes as are requested at the time of declaration.
2. Pointers are 'agnostic'; they don't need to know what they're pointing to. By contrast, an array is always an array of *something* (ints, chars, floats, structs…)
3. For a pointer to be useful, it must point to something which has been allocated by the operating system. Arrays do not suffer from this problem; they automatically occupy the space they are assigned to.
4. Arrays are restricted to the location at which they are found; pointers, by contrast, may point to any location in RAM. Hence the address of the start of an array is a constant.
5. Arrays are not passed to functions, except in the form of pointers
6. Pointers are capable of pointer arithmetic; this feature is not available in arrays (except that the starting element of the array may be used as an address in a pointer)

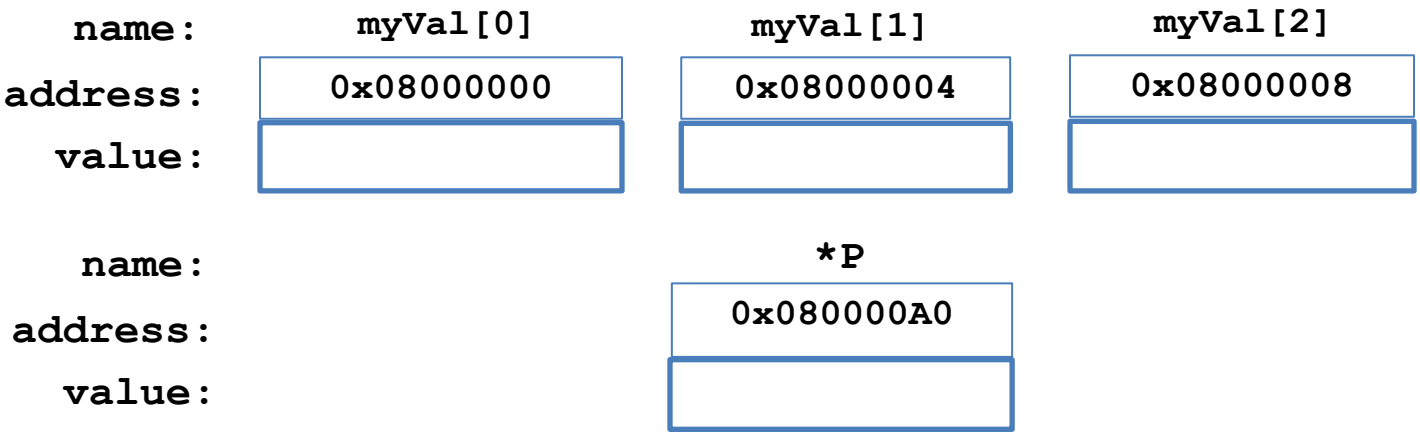
/6

34. Given the declaration:

```
int myVal[3] = {1, 2}, *P = myVal;
```

(a) indicate which values would go in each of the four empty boxes shown below.

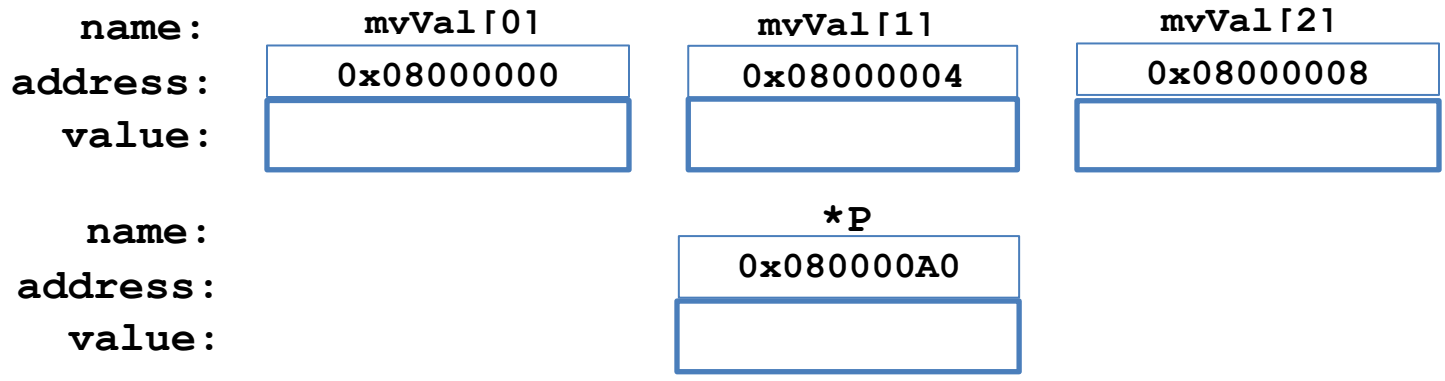
(b) indicate how `P` is initialized by drawing an arrow from the appropriate location in the `myVal` array to `P`; in other words, show how the pointer gets its value.



(c) Assume we now execute the statements:

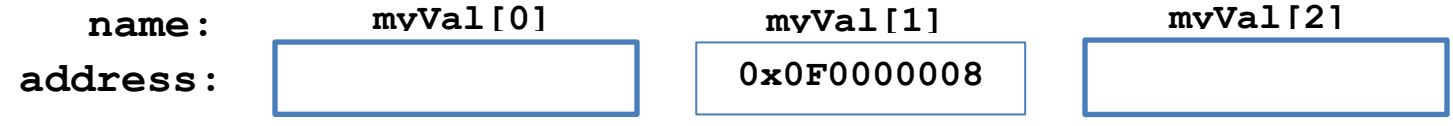
```
P++;
*P  = *(P - 1);
```

indicate in the boxes below the new values



(d) Assume the original declaration above was for `double` values (i.e. an 8-byte data type) rather than `int`s. Fill in the boxes assuming this was the original declaration, using the address for `myVal[1]` as a guide.

```
double myVal[3] = {1.0, 2.0}, *P = myVal;
```

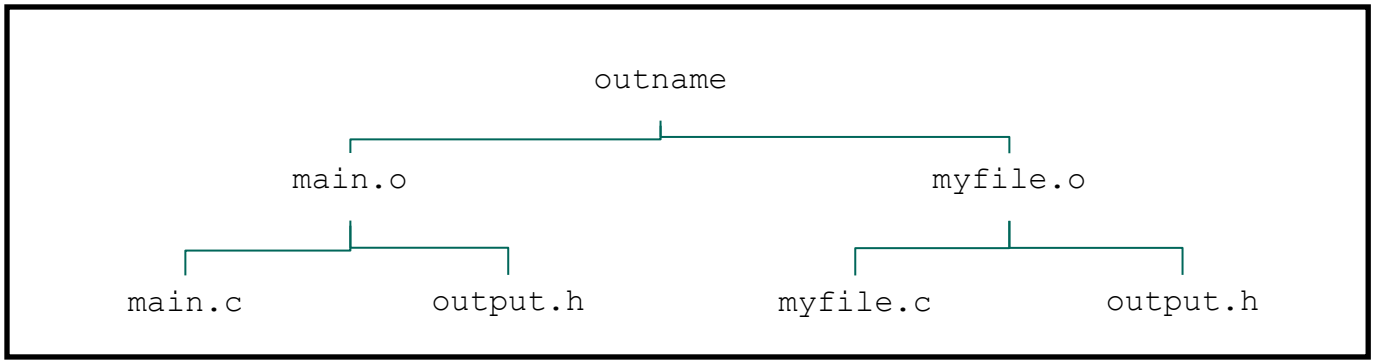


/10

35. Shown below is a makefile for a program involving three files, `main.c`, `myfile.c`, and `output.h`. The latter file, `output.h`, is assumed to be included in both `main.c` and `myfile.c`. In the box below, draw the dependency tree:

```
CC = gcc                              # define gcc as CC
CFLAGS = -ansi -pedantic -Wall        # set compiler flags
OBJS = main.o myfile.o                # set obj files to OBJS
PROG = outname                        # set outname to PROG

$(PROG): $(OBJS)
   $(CC) $(OBJS) -o $(PROG)
main.o: main.c output.h
   $(CC) $(CFLAGS) -c main.c
myfile.o: myfile.c output.h
   $(CC) $(CFLAGS) -c myfile.c
```



36. The following should be printed on the last page of the exam, following question 38 **/6**

(a) Write a function called `sumASCII()` that (1) takes as an argument a string and (2) iterates through each character of the string and calculates the ASCII equivalent value of the entire string, and then (3) divides through by the total number of characters in the string.

(b) Also write a `main()` function that calls `sumASCII()` and displays the output returned from the function.

(c) Finally, write a forward declaration for `sumASCII()`.

```
#include <stdio.h>
float sumASCII(char * string);

int main(void){
   char str[80];
   printf("Enter a string\n");
   scanf("%s", str);
   printf("The average ASCII value of the string is %f", sumASCII(str));
}
float sumASCII(char * string){
   int tot=0, n=0;
   while (*string){
      tot += *(string++);
      n++;
   }
   return ((float)tot/n);
```

/6

37. The following code was written with the intention of (1) passing a fixed string to a function, which then (2) counts the total number of uppercase characters in the string, (3) prints out the presumed number of lowercase (LC) characters based on the length of the string and the number of uppercase characters, and then (4) returns a pointer to `main()` containing the uppercase characters detected in the string. Unfortunately the code contains a number of errors, both compile-time and run-time. Place a number next to these errors in the code below, and explain why the code is in error in the box at right next to the number you used in the code.

```
#in   de <std  >
char getUCaseChars(char);

int main(void){
   char Ar[] = "ABCdefghijKLM";
   printf("Uchars in string are : "
      "%c\n", getUCaseChars(&Ar));
}

char *getUCaseChars(char ar[]){
   int totUCase = 0, i = 0;
   char UCStr[80]  {0};
   while (!*ar){
      if (isupper(ar)){
         totUCase++;
         UCStr[i++] = *ar
      }
      ar++;
   }
   printf("Total number of LC chars is %d\n", (sizeof(ar) - totUCase));
   return (&UCStr);
}
```

Here's the correct, fully functioning program (although note that it uses sizeof() incorrectly, so it still prints out the wrong number of LC characters.)

```
#include <stdio.h>
#include <ctype.h>

char *getUCaseChars(char []);

int main(void){
   char Ar[] = "ABCdefghijKLM";
   printf("Uchars in string are:"
      "%s\n", getUCaseChars(Ar));
}

char *getUCaseChars(char ar[]){
   int totUCase = 0, i = 0;
   static char UCStr[80] = {0};
   while (*ar){
      if (isupper(*ar)){
         totUCase++;
         UCStr[i++] = *ar;
      }
      ar++;
   }
   printf("Total number of LC…)
   return (UCStr);
}
```

You may assume that (1)) you may assume that only alphabetic characters are entered into the input string (2) the fundamental construction of the code is essentially correct even if its implementation is unnecessarily bloated and flawed in many critical areas. Your job is not to critique the entire construction of the code, but simply to find *specific* compilation and run-time/logic errors. (3) If you find an error that seems to occur more than once, chances are that that code is in fact correct, and the real error lies elsewhere in another location, and only once. Even if an error does occur more than once, you only get credit for the first occurrence of that error, not for repeat errors of the same kind.