CS240 Final Solution, summer 2023

P1(a) 15 pts

x occupies 8 bytes in main memory: 4 bytes for the first field, int a,
and 4 bytes for the second field, unsigned int b.
3 pts

y occupies 4 bytes in main memory: the same 4 bytes can be used to stored
the value of the first field or the value of the second field, but not both.
3 pts

If y only uses its first field, y.a, or second field, y.b, but not both at
the same time, union may be used and it conserves memory.
3 pts

union is not appropriate if the values of both fields need to be preserved.
3 pts

200. Since both fields share the same 4-byte memory location y.b = 200 overwrites
the previous value 100.
3 pts

P1(b) 15 pts

fgetc() communicates to the calling function that the end of file has been
reached by returning -1. To do so, it needs more than 1 byte since all 8 bits
of a byte returned by fgetc() if the end of a file is not reached are used
to represent the 1 byte data read.
7 pts

printf() will likely output -21.123456xyz where xyz are "random" digits, or
-21.1234567xy.
4 pts

This is so since float can handle precision up to 6-7 digits below decimal point.
4 pts

P2(a) 15 pts

mysum() used a format string similar to printf() where upon detecting symbol '%'
the next character was checked to determine if it was 'd' (integer), 'c' (character),
's' (string), etc.
4 pts

scanf() checks for occurrences of '%' in the format string (ignoring %% which
prints the symbol '%') where each occurrence indicates that an additional argument
should follow.
5 pts

printf("%f",u,v) will output the value of u (and ignore v). This is so since
printf() is coded to look for occurrences of '%' which are then matched to an
argument by calling va_arg().
3 pts

printf("%f %f",u) will lead to an additional value being output for the second
%f due to calling va_arg() twice. The second output will be a junk/random value
since an argument for the second %f was not passed.
3 pts

P2(b) 15 pts

const specifies that the function dosomething() will not modify the value of
the pointer argument. gcc will generate an error if the code of dosomething()
attempts to do so.
8 pts

No guarantees are provided.
3 pts

For example, in example code

```
int dosomething(const char *m) {
int *x;
  x = m;
  *x = '\0';
}
```

will modify a string (or sequence of characters) pointed to by
argument m to be an empty string. gcc will provide a warning but
compile the code.
4 pts

P3(a) 20 pts

```
int main(int argc, char **argv) {
  // 2 pts

int i, x, y;

  // Basic sanity check.
  if(argc == 1) {
          printf("Too few arguments\n");
          exit(1);
  }
  // 3 pts

  // Initialization.
  y = 1;
  // 2 pts

  // When looping exclude argv[0] which is the command itself.
  for(i=1; i<argc; i++) {
  // 6 pts

  // Conversion from string to integer.
          x = atoi(argv[i]);
  // 4 pts

          y = y * x;
  }
  return y;
  // 3 pts
}
// In the above we are not performing advanced sanity checks such as the
// command-line arguments not being strings that represent numbers.
// This can be done by checking the return value of atoi() which is not
// necessary.
```

P3(b) 20 pts

```
int main(int argc, char **argv) {
char buf[100];
// 2 pts

int numlines, x, y, i, j;
char **stringarray;
FILE *fp;

  fp = fopen(argv[1],"r"); // Omitting check if fopen() failed.
  numlines = atoi(argv[2]);
  // 2 pts

  // Allocate 1-D array of pointers to strings.
  stringarray = malloc(numlines * sizeof(char *));
  // 4 pts

  i = 0;
  j = 0;
  while((x = fgetc(fp)) != EOF) {
  // 2 pts
          buf[i] = (char) x;    // Type conversion (char) may be omitted.
```

```
            if((buf[i] != '\n') {
              i++;
              continue;
            }
            // 2 pts

            // Make read line into string by overwriting newline with EOS.
            buf[i] = '\0';
            // 2 pts

            // For each pointer in the 1-D array allocate memory for line read from file.
            *(stringarray + j) = malloc(i * sizeof(char));         // Can be just malloc(i).
            // 4 pts

            // Copy read string into corresponding 1-D array of string pointers.
            strcpy(*(stringarray + j), buf);
            // 2 pts

            i = 0;
            j++;
  }

}
```

Bonus 10 pts

q occupies 4 bytes.
4 pts

q.x0 values: 0 or 1 since single bit
q.x1 values: 0, 1, ..., 15 since 4 bits
q.x2 values: 0, 1, 2, 3 since 2 bits
6 pts