1. (a) A parent process and a child share the same address space.

☐ True ☒ False

(b) A process can call wait to wait for a process that it is not related to.

☒ True ☐ False

(c) Signals can be used for communication between processes on different machines.

☒ True ☐ False With scokets maybe?

(d) Internet routers guarantee reliable delivery of network packets.

☐ True ☒ False TCP ensures reliability

(e) TCP is more efficient, but less reliable than UDP.

☐ True ☒ False UDP is more efficient

(f) A string's length (as indicated by strlen) and its size (as indicated by sizeof) are always equal.

☐ True ☒ False

(g) Pipes can be used for communication between processes on different machines.

☒ True ☐ False With named pipes

(h) Sockets can be used for communication between processes running on the same machine.

☒ True ☐ False

2. (a) Considering the following piece of code, fill the table below with the values of the array elements after the code is done executing. Be careful with the difference between pointers and values, and pointer arithmetic.

```
int a[4] = {0, 1, 2, 3};
int b = 1;
int *p = a;
p = p + b;
b++;
*p += b;
p = p + b;
*p += 2;
p--;
*p *= 4;
p = p - b;
*p = p - a;
```

a[0]	a[1]	a[2]	a[3]
0	3	8	5

(b) What is the output of the following program?

```
#include <stdio.h>

int func(int a, int *b, int *c) {
  a += 5;
  *b += a;
  c = b;
  return a;
}

int main() {
  int x = 5, y = 8, z = 3, t = 0;
  t = func(x, &y, &z);

  printf("x:%d    y:%d    z:%d  t:%d\n", x, y, z, t);
  return 0;
}
```

```
x:5  y:18  z:3  t:10
```

(c) There are several errors in the following program. Use the appropriate letter to label the lines of code where the corresponding error occurs.

P - dereferencing a pointer without having allocated memory for it.

D - deallocating memory that has already been deallocated.

H - deallocating memory that is not located on the heap.

M - memory leak.

```
int a = 0, b = 1, c = 2;
int *p, *q, *r;

p = malloc(sizeof(int));
*p = b;
q = &c;
p = &a;     (M, since p now has address of a, malloc'd memory has no pointer to it)
*q = b;
q = malloc(sizeof(10));
*r = a + b + c;    (P)
free(p);  (H, p points to a)
r = q;
p = malloc(sizeof(int));
*p = *r;
free(q);
free(r);  (D, r and q point to same thing)
free(p);
```

(d) What does the following piece of code print?

```
struct Point {
  int x;
  int y;
};

struct Point p1, p2;
struct Point *q1, *q2;

p1.x = 5;  p1.y = 10;
p2.x = 1;  p2.y = 10;

q1 = &p2;
q2 = &p1;
q1->x += 2;    q1->y += 7;
q2->x += 4;    q2->y += 3;

printf("P1(X,Y) = (%d, %d)\n", p1.x, p1.y);
printf("P2(X,Y) = (%d, %d)\n", p2.x, p2.y);
```

```
P1(X,Y) = (9, 13)
P2(X,Y) = (3, 17)
```

3. Each example below contains an independent code fragment. In each case, there are variables x and y that are missing declaration statements. In the boxes to the right of the code, write those declaration statements so that the code fragment would compile and run without warnings or errors. If there is no declaration that could lead to a compilation without warnings or errors, write "ERROR". The first is done for you as an example.

Code Fragment	**Declaration for x**	**Declaration for y**
`x = 10;` `y = 'A';`	`int x;`	`char y;`

`double length = 25;`
`x = &length;`
`y = &x`
	`double *x;`	`double **y;`

`char *id[6];`
`x = id[3];`
`// some hidden code`
`id[3] = "c3new";`
`y = *x[3];`
	`char *x;`	`char *y;`

`char *name = "John Tory";`
`x = &name;`
`y = *(name + 3);`
	`char **x;`	`char y;`

`struct node {`
`  int value;`
`  struct node *next;`
`};`
`typedef struct node List;`
`List *head;`
`// some hidden code`
`x = head->next;`
`y.value = 14;`
`y.next = x;`
	`struct node *x;`	`struct node y;`

`char fun(char *str, int n) {`
`  return str[n];`
`}`
`y = fun("hello", 1);`
`x = &y;`
	`ERROR: fun has wrong return type`	`ERROR: fun has wrong return type`

4. Some of the code fragments below have a problem. For each fragment indicate whether the code works as intended or there is an error (logical error, compile-time error/warning, or runtime error). Assume all programs are compiled using the *C*99 standard. For this question, we will assume programs which do not terminate are errors as well. If there is an error in a fragment, explain **briefly** what is wrong in the box. For parts where there is an error you will only receive marks if you can correctly explain what is wrong. We have intentionally omitted the error checking of the system calls to simplify the examples. Do **not** report this as an error.
Some of the parts use the following struct definition:

```
struct student {
  int age;
  char *name;
};
```

```
char *s = "Hello";
strcat(s, ", World!");
```

☐ Works as intended ☒ Error
`*s` is declared as a string literal and therefore cannot be modified.

```
int main(int argc, char **argv) {
  char ch;
  char *p = &ch;
  ch = argv[argc-1][0];
  printf("%c\n", p[0]);
  return 0;
}
```

☒ Works as intended ☐ Error
Prints the first char of `argv[0]`, i.e. first char of the program name.

```
struct student hannah;
strcpy(hannah.name, "Hannah");
```

☐ Works as intended ☒ Error
No memory allocated for `hannah.name`.

```
struct student hannah = NULL;
// ... missing code ...
if (hannah != NULL) {
  hannah.age = 10;
}
```

☐ Works as intended ☒ Error
`NULL` can only be assigned to pointers, so type mixup.

```
// Increase the age of a student by amt.
void increase_age(struct student s, int amt) {
  s.age += amt;
}

int main() {
  struct student rob;
  rob.age = 10;
  increase_age(rob, 5);
  printf("%d should be 15\n", rob.age);
  return 0;
}
```

☐ Works as intended ☒ Error
`age` changed in `increase_age` locally, but change will not be reflected in `main`. `increase_age` should use pointers instead.

```
// Compute the sum of an array of integers
int compute_sum(int numbers[]) {
  int sum = 0;
  for (int i = 0; i < sizeof(numbers); i++) {
    sum += numbers[i];
  }
  return sum;
}
```

☐ Works as intended ☒ Error
`sizeof(numbers)` in `compute_sum` will return the size of an `int *`, which will not loop over the array successfully. An additional parameter should be added to `compute_sum`, that explicitly has the number of elements in `numbers`.

```
int fd[2];
int result = fork();
pipe(fd);
if (result == 0) {
  close(fd[0]);
  write(fd[1], "cscb09", 7);
}
else {
  close(fd[1]);
  char buf[7];
  read(fd[0], buf, 7);
  printf("%s\n", buf);
}
exit(0);
```

☐ Works as intended ☒ Error
`pipe(fd)` should be above the `fork()` line, so that both parent and child inherit the same fds.

```
// Read all bytes from the file descriptors in ‘fds’ as characters,
// and print them. ‘num_fds’ is the number of file descriptors, and
// ‘max_fd’ is the value of the largest one.
void read_ints(int *fds, int num_fds, int max_fd) {
  char data;
  fd_set set;
  FD_ZERO(&set);
  for (int i = 0; i < num_fds; i++) {
    FD_SET(fds[i], &set);
  }
  while (select(max_fd + 1, &set, NULL, NULL, NULL) > 0) {
    for (int i = 0; i < num_fds; i++) {
      if (FD_ISSET(fds[i], &set)) {
        if (read(sum, &data, 1) > 0) {
          printf("%c\n", data);
        }
      }
    }
  }
}
```

☒ Works as intended ☐ Error

```
struct node {
  int item;
  struct node *next;
};

// Compute the sum of the items in a linked list with the given head,
// but do not modify the list.
int sum(struct node *head) {
  int s = 0;
  while (head != NULL) {
    s += head->item;
    *head = *(head->next);
  }
  return s;
}
```

☐ Works as intended ☒ Error

`*head = *(head->next)` is wrong. This makes the memory that `head` points to have the same contents that `head->next` points to. Also, this modifies the list by modifying `head`, which means the actual `head` cannot be accessed again. A `temp` pointer should be used to traverse list instead.

```
// Remove the dots from word
char *word = "Ex.ampl.e";
char *result = malloc(strlen(word) + 1); // upper-limit if word has no dots
for (int i = 0; i < strlen(word); i++) {
  if (word[i] != '.') {
    strncat(result, word[i], 1);
  }
}
```

☐ Works as intended ☒ Error

`strncat` uses `char` pointers as arguments so `strncat(result, &word[i], 1);` should be used instead.

5. Assume a linked list structure, which contains some information about a student, as follows:

```
typedef struct student {
  int student_number;
  char *last_name;
  struct student *next;
} Student;
```

Write a function that traverses a given student list and inserts a new student in alphabetical order (function prototype given below). If the new student has the same last name as another student from the list, insert in ascending order of the student number.
Note: The new student name passed to the function may be deallocated once the function exits, so make sure to create a new copy of the student name when inserting.

```
Student *insert_new_student(Student *list, int newStudentNo, char *newName) {
    if (list == NULL) {
        Student *node = (Student *) calloc(1, sizeof(Student));
        node->last_name = (char *) calloc(strlen(newName), sizeof(char));
        strcpy(node->last_name, newName);
        node->student_number = newStudentNo;
        node->next = NULL;
        return node;
    }
    for (Student *temp = list; temp != NULL; temp = temp->next) {
        if ((strcmp(temp->last_name, newName) < 0)) {
            Student *node = (Student *) calloc(1, sizeof(Student));
            node->last_name = (char *) calloc(strlen(newName), sizeof(char));
            strcpy(node->last_name, newName);
            node->student_number = newStudentNo;
            node->next = temp->next;
            temp->next = node;
            return list;
        }
    }
}
```