1. (10) (Warm up) Given the C variables defined below, for each of the following expressions, give its value.

```
int   A = 3;     char   *cp = "acebdf";
int   B = 2;     int    grade[6] = { 4, 2, 1, 0, 1, -1 };
```

If it is relevant, assume that `sizeof(int)` is 4, `sizeof(char)` is 1, and `sizeof()` any pointer type is 4. Treat each expression as independent.

| Expression | Value |
|---|---|
| `A - B` | (int) 1 |
| `B / A` integer div | (int) 0 |
| `++A + ++B` | (int) 7 |
| `*(grade + 2)` | 1 |
| `cp[3]` | b |
| `&grade[4] - &grade[0]` → grade+4, grade+0 | (grade+4) - (grade+0) → → 4 |
| `(int)&grade[4] - (int)&grade[0]` | ~~(int) 4~~ let int = 4; (int) x(offset) → (4)(4) → 16 ÷ 4 |
| `(A=1)?B:A` | ~~3~~ assigning to truthy value satisfied the ternary operator meaning this expression returns 2 |
| `strlen(cp)` | 6 |
| `sizeof(cp)` | (1)·(6) → ~~6~~ a ptr is usually a 8 byte value so it should be 8 here. |

2. (8) Given the following definitions:

```
#define twice(x) (x) + (x)
#define nonce(x) (3)
int  x = 6;
int  y = 7;
```

what is the value of each of the following expressions:

| Expression | Value | Explanation |
|---|---|---|
| `twice(2)` | 2 | we are replacing x in twice with 2 the eval. expression (2)+(2) |
| `twice(y++)` | 14 | we are placing y in the macro then incrementing its value by 1. (7)+(7) |
| `twice(2) * twice(2)` | ~~12~~ 8 | (2)+(2)*(2)+(2) → 8 multiply first |
| `twice(nonce(x))` | UDF ~~6~~ | this is undefined as macro is a text replacement. twice(nonce(x)) → → nonce(x) + nonce(x) → → 3+3 → 6 |

5

3. (4) What is the meaning of an *extern* declaration in C?

it means the variable with thes declaration can be called from any where as long as you include the file that first init. it Example getopt(); have the extern variable optarg that can be called from your own file as long as you include <getopt.h>. TR:DL: extern is a global variable that can be implicitly call.

1

- variable defined outside current file

close enough :)

4. (8) Implement a C function `CountOdd()` that takes an array of integers `A` and its length `len` and returns the number of odd integers in `A`. You may assume that you will receive proper arguments.

```
int CountOdd(int A[], int len){
  int i, count;        i = 0;
  while(i<len)         count = 0;
  {
    if((A[i] % 2) != 0) || A[i] != 0)
    {
      count++;
    }
    i++;
  }
  return count;
```

!!! this is only fix for 0

6

```
int CountOdd(int A[], int len)
{
  int i, count;
  i = 0;
  count = 0;
  while(i<len)
  {
    if((A[i] % 2) != 0) && A[i] != 0)
    {
      Count ++;
    }
    i++;
  }
  return count;
}
```

needed to change

5. (15) Implement a robust version of the C library function strrchr(3):

**Name:**
`char *strrchr(const char *s, int c);`

**Description:**
The strrchr() function returns a pointer to the last occurrence of the character c in the string s.

**Return Value:**
strrchr() function returns a pointer to the matched character or NULL if the character is not found. The terminating null byte is considered part of the string, so that if c is specified as '\0', this function returns a pointer to the terminator.

Write robust code (even though the library version is fragile). That is, return NULL on failure, but do not crash. Do not use any of the C library's string functions. Think before you write anything.

```
char *donde_es_char(const char *S, int c)
{
    char *str;
    str = NULL;
    str = strrchr(S, c);
    return str;
}
```

I don't know how this function can crash given the arguments above.

er, calling strrchr() to implement strrchr() isn't really in the spirit of the thing. It is a function, I guess

```
Char *strrchr(const char *S, int c)
{
    char *ch, *rt_ch;
    ch = S;
    rt_ch = NULL;
    while (*ch)
    {
        if (*ch == c)
        {
            rt_ch = ch;
        }
        ch++
    }
return rt_char;
}
```

gotta read the question carefully :)

6. (15) Given the following structure definition:

```
struct node_st {
    int data;
    struct node_st *next;
};
```

Write a C function, `sorted_insert_list()`, that takes an integer argument, `num`, and a linked list, `list`, (possibly NULL) of these structures sorted in ascending order. `sorted_insert_list()` creates a new node with the given value and inserts it in the list so that all the data values in the list remain sorted. `sorted_insert_list()` returns a pointer to the head of the new list.

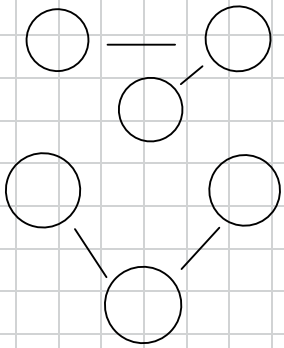
Write robust code.

```
struct node_st *sorted_insert_list(int num, struct node_st *list){
```

```
struct node_st *node, *ndata;
node = list;
ndata->data = num;
if (node != NULL)
{
while (node->next != NULL)
{
  if (node->next->data > num)
  { ndata->next = node->next;
    node->next = ndata;
  return node;
  }
  node = node->next;
}
ndata->next = NULL;
return ndata;
```

```
// head node
if (node->data > num)
{
ndata->next = node;
return ndata;
}
```

```
node->next = ndata;
ndata->next = NULL;
return node;
```



```
struct node_t *sorted_insert_list(int num, struct node_st *list)
{
 struct node_st *node = list;
 struct node_st *ndata;
 ndata = (struct node_st*)malloc(sizeof(struct node_st));
 ndata->data = num;
 ndata->next = NULL;
if(!node)
{
 while((node->next))
 {
  if(node->next->data > ndata->data)
  {
   ndata->next = node->next;
   node->next = ndata;
   return node;
  }
  node = node->next;
 }
 ndata->next = NULL;
 return ndata;
}
if(node->data > ndata->data)
{
 ndata->next = node;
 list = ndata;
 return list;
}
node->next = ndata;
return list;
```