1. What is the output of the following program? Explain.

```
#include <stdio.h>

int main(){
     int val[] = {5,5};
     int *ptr = val;
     printf("%d \n", ++*ptr);
     printf("%d \n", *++ptr);
     return 0;
}
```

Have the elements of val[] changed? Which one or both?

Answer:

Output:
6
5

The first `printf` statement increments by 1 the content of memory location in `ptr` which is the first element of array `val` or `val[0]`.

The second `printf` statement increments the pointer value by one element `++ptr` then gets the content, which is the content of `val[1]`.

At the end of the function, `val[0]`'s original content has changed to 6 while `val[1]` remains unchanged.

2. Why should pointers have data types when their size is always the same, e.g. 4 bytes (in a 32-bit machine), irrespective of the variable they are pointing to?

Answer:

i) Allows the compiler to spot potential incorrect operations on the contents of the memory location pointed to by the pointer.

ii) For an array, consecutive memory is allocated. Each element is placed at a certain offset from the previous element, if any, depending on its size. The compiler that generates code for a pointer, which accesses these elements using the pointer arithmetic, requires the number of bytes to retrieve on pointer dereference and it knows how much to scale an array index. The data type of the pointer provides this information. The compiler automatically scales an index to the size of the variable pointed at. The compiler takes care of scaling before adding the base address.

3. Given the following program which compiles without error:

```
int main(){

      int p, *ip;
      float q, *fq;
      double r, *dr;

      ip = &p;
      fq = &q;
      dr = &r;

      *fq=22.0/7;
      printf("1: %f\n", *fq);

      ip=(int*)fq;
      dr=(double *)fq;

      printf("2: %d\n", *ip);
      printf("3: %lf\n", *dr);

      return 0;
}
```

Explain the outputs produced by executing the program multiple times, as shown below:

```
$ ./a.out
1: 3.142857
2: 1078535314
3: 88855742519939419180832258233417179953855827761765859111951058848631573706702848.000000

$ ./a.out
1: 3.142857
2: 1078535314
3: 124110321621185987661417250795552426884588450161101694201232754853357430760317911104.000000

$ ./a.out
1: 3.142857
2: 1078535314
3: 594445518536611055432452280412104513591252516566931558459288325106216387781591104.000000
```

Hint: why is it consistently "2: 1078535314" while "3: …" is changing each time?

Answer:

`fq` is a pointer to a floating point variable `q` which stores the value of π or 22/7 in floating point format.

When `fq` is assigned to `ip`, a pointer to an integer, `*ip` interprets the contents of the memory location pointed to by `ip` as an integer, i.e. the sign-exponent-mantissa format now assumed to be simple binary format. Since both integer and single-precision floating point variable occupy the same number of bytes, the output is wrong but consistent.

`dr` is a pointer to a double-precision floating point variable of size 8 bytes. The first 4 bytes are the same in each execution, but the next four bytes are undefined since they are not allocated. Hence, each time `printf` tries to print the contents pointed to by `dr`, there are different contents in the next 4 bytes, hence the unpredictable output.

4. If global variables can be used anywhere in the program, why not make all variables global? (Hint: remember where global variables are stored, i.e. which segment and for how long, and similarly for local variables.)

Answer:
Variables declared as global take up memory for the entire time during which the program runs; however, local variables do not. A local variable takes up memory only while the function to which it is local is active. When a program becomes complex and large, it may be needed to declare more and more variables, taking up lots of memory that are not likely to be used all the time.

Additionally, global variables are subject to unintentional alteration by other functions. If this occurs, the variables might not contain the values one expects when they are used in the functions for which they were created.

5. What is the difference between an **internal** static variable and an **external** static variable?

Answer:
An internal static variable is declared inside a block with static storage class whereas an external static variable is declared outside all the blocks in a compilation unit (i.e. file.) An internal static variable has persistent storage, block scope, and no linkage. An external static variable has permanent storage, file scope, and internal linkage.