Complicated Declarations:

1) Declare an integer.
   Choose a name for variable. Say 'n'
   int n;

   keyword int is known as type annotation.

2) Declare a pointer to integer.

   Select a name: 'p'
   How to declare that p is a pointer? -> *p
   How to declare that p is a pointer to int?
   int *p;

3) Declare an array 5 of integers.
   Select a name: arr
   How to declare that arr is an array?
   arr[]
   How to specify a size of array? How to specify that arr is an array of
   five elements? arr[5]
   How to specify that the type of array element is int.

   int arr[5];

4) Declare a function accepting integer and returning integer.

   Select a name: test_func
   How to declare that test_func is a function? test_func()
   How to declare that the test_func accepts an integer?
   test_func(int);
   How to declare that the test_func returns an integer?
   int test_func(int);

5) If struct T is a user defined data type then its instance can be declared
   as follows:
   struct T inT;
   and pointer to struct T is defined as follows:
   struct T* pT;

Observations:

0) We can declare either a) an instance of built in data type b) instance of
user defined data type c) array of built in data types d) array of user defined

data type e) array of pointer to built in types f) array of pointer to user defined data type g) function with any formal parameter list and return type

f) pointer to built in data type
   pointer to user defined data type
   pointer to array
   pointer to function
   pointer to pointer

1) While declaring an instance of built in data type or user defined data type the entire type annotation falls on LHS of the variable name.

2) While defining array of any element type and while defining any function the type annotation (= type information) falls on both sides of variable name.

3) While declaring a pointer, the pointer syntax, that of dereference operator ('*') falls on LHS of the variable. If pointer to any entity in 1) is being declared then entire type information falls on LHS. If pointer to any entity in 2) is being declared then the type information falls on LHS and RHS.
   If type information of entity falls entirely on LHS then
   type information while declaring a pointer to such entity also falls on LHS

   If type information of entity falls on LHS & RHS then
   type information while declaring a pointer to such entity also falls on LHS & RHS

#--

```
int n;
int *p;
```

POINTER TO BUILTIN DATA TYPES

```
char* p; unsigned char* p; short* p, unsigned short* p; int* p; unsigned int* p;
long* p; unsigned long* p; long long *p; unsigned long long* p; float* p;
double* p; long double* p;
```

POINTER TO USER DEFINED DATA TYPES

```
struct Book* pbook; struct Date* pDate; struct Pen* pPen; struct student* pSt;
```

POINTER TO ARRAY

```
int arr[5];  // syntax to denote that arr is array falls on RHS of varaible name
             // syntax to denote array element type falls on LHS of vname.

int (*p) [5];
```

POINTER TO FUNCTION

```
int func(int a, int b);

int    (*pfn)  (int, int) ;
```

POINTER TO POINTER:

```
int* p;
int** pp;
```

```
#-----------------------------------------------------------------------------
```

```
int n;
```

1) n is
2) n is an integer.

```
struct student s;
```

1) s is.
2) s is an instance of struct student

```
int arr[5];
```

1) arr is
2) arr is array
3) arr is array 5 of
4) arr is array 5 of int.

```
int test(int, int);
```

1) test is
2) test is a function
3) test is a function accepting an int and int
4) test is a function accepting an int and int and returning int.

```
int* p;
```

1) p is
2) p is a pointer
3) p is a pointer to int.

```
struct student* ps;
```

1) ps is
2) ps is a pointer
3) ps is a pointer to struct student

```
int (*pa) [5];
1) pa is
2) pa is a pointer
3) pa is a pointer to array
4) pa is a pointer to array 5 of
5) pa is a pointer to array 5 of int.

int (*pfn) (int, int);
1) pfn is
2) pfn is a pointer to
3) pfn is a pointer to a function
4) pfn is a pointer to a function accepting an int and int.
5) pfn is a pointer to a function accepting an int and int and returning int.

Rule:
   1) You must DEFINE only ONE VARIABLE NAME IN ONE DATA DEFINITION STATEMENT.
   int (*pfn)(int a, int b);

struct A {
   int a;
   char b;
   float c;
};

int func(int x, int y)
{
   double m, n;
}

#-------------------------------------------------------------------------

1) Declare a function accepting
   @P1) an integer
   @P2) pointer to a function
                        @P1) int
                        @return: void
   @return: pointer to a function
                        @P1) int
                        @return: void

Declare a function accepting an integer and a pointer to a function
accepting integer and returning void
returning pointer to a function accepting int and returning void.
```

1) test
2) test()
3) test(int)
4) test(int, (*)()

#---------------
How to go about pointer to function
(*p)();
(*p)(formal parameter list)
return_type (*p)(formal parameter list);
#---------------
HOW TO DROP FORMAL PARAMTER NAMES WHILE DECLARING FUNCTIONS

```
int test(int a, int b)
{

}

int test(int, int);

void sort(int*, int);

void sort(void* arr, int N, int nmem, int (*compare)(const void*, const void*));

void sort(void*, int, int, int (*) (const void*, const void*));
```

#---------------------

RESTARTING ORIGINAL PROBLEM

1) Declare a function accepting
   @P1) an integer
   @P2) pointer to a function
                @P1) int
                @return: void
   @return: pointer to a function
                @P1) int
                @return: void

1) test
2) test()
3) test(int)
4) test(int, void(*)(int))

```
5)  (*)()  test( int, void(*)(int) )
6) void(*)(int)  test( int, void(*)(int) ); // logically correct
                                            // syntax requires adjustment
```

sub-example:

Declare a function accepting int and returning a pointer to int

```
1) test
2) test()
3) test(int);
4) *test(int);
5) int* test(int);
```

sub-example:

Delcare a function accepting int and returning pointer to array 5 of int.

```
1) test
2) test()
3) test(int)
4) *test(int)
5) *test(int) [5]
6) int *  test(int) [5]; // This still requires correction. (PRECEDENCE
CORRECTION)
```

```
1) test
2) test()
3) test(int)
4) *test(int)
5) *   test(int) [5];
6)  (*test(int))[5];
7) int (*test(int))[5];
```

RESTARTING ORIGINAL PROBLEM

```
1) Declare a function accepting
   @P1) an integer
   @P2) pointer to a function
                        @P1) int
                        @return: void
   @return: pointer to a function
                        @P1) int
                        @return: void
```

1) test
2) test();
3) test(int)
4) test(int, (*)());
5) test(int, void(*)(int))
6) *test(int, void(*)(int))
7) (*test(int, void(*)(int)))()
8) void (*test(int, void(*)(int)))(int);

void (*   test(int, void(*)(int))   )(int);

void (*   pfn   )(int);

void (*) (int);

[
   Unix is a simple operating system. But it takes genius to understand its
   simplicity.
]

---

1) Declare a pointer to a function accepting int returning pointer to
   array 5 of pointer to functions accepting int and returning int

   Declare a pointer to function
      @p1: int
      @return: pointer to array 5 of pointer to function
                                          @p1: int
                                          @return: int

1) pfn
2) (*pfn)()
3) (*pfn)(int)
4) (*(*pfn)(int))[5]
5) (*(*(*pfn)(int))[5])()
6) int (*(*(*pfn)(int))[5])(int);

```
Declare pointer to array 5 of pointer to function
    @p1: int
    @p2: pointer to a function
           @p1: int
           @p2: pointer to array 5 of integers
           @return: pointer to pointer to int
    @return: pointer to function
               @p1: int
               @return: pointer to array 3 of pointer to function
                          @p1: int**
                          @p2: int**
                          @return: void**
```

1) pa
2) (*pa)[5]
3) (*(*pa)[5])()
4) (*(*pa)[5])(int, int** (*)(int, int(*)[5]))
5) (*(*(*pa)[5])(int, int** (*)(int, int(*)[5])))()
6) (*(*(*(*pa)[5])(int, int** (*)(int, int(*)[5])))(int))[3]
7) (*(*(*(*(*pa)[5])(int, int** (*)(int, int(*)[5])))(int))[3])()
8) void**(*(*(*(*(*pa)[5])(int, int** (*)(int, int(*)[5])))(int))[3])(int**, int**);
