Solution to Stage Question

Question 1.

Memory partitions when C programs run. Such as code, data, bss, heap, stack.
Please consult the information to understand it.

When a C program runs, it utilizes different memory partitions to store different types of data and code. These memory partitions include code segment, data segment, BSS (Block Started by Symbol) segment, heap, and stack. Below is the partitions of memory in C.

i. Text segment (i.e. instructions)
ii. Initialized data segment
iii. Uninitialized data segment (bss)
iv. Heap
v. Stack

Fig. 1: A typical memory layout of a running process.

Here is the discussion of each of them:

i.Code Segment(instructions): The code segment, also known as the text segment, is where the compiled program instructions are stored. It contains the executable instructions of the program. The code segment is typically read-only and shared among multiple instances of the same program. It is loaded into memory when the program starts and remains constant during program execution o prevent a program from accidentally modifying its instructions.

Example:

```
#include <stdio.h>
int main() {
  printf("Hello, World!");
  return 0;
}
```

In the above example, the instructions to print "Hello, World!" are stored in the code segment.

Data Segment:

The data segment contains global and static variables that are initialized with a specific value. This segment is divided into two parts: Initialized data and Uninitialized data.

ii.Initialized Data Segment:

The initialized data segment, also known as the data segment, is a part of a program's virtual address space that stores global variables and static variables. These variables are initialized by the programmer and can be modified during runtime. The data segment can be further divided into two areas: initialized read-only and initialized read-write.

In the initialized read-write area of the data segment, variables such as global strings (e.g., char s[] = "hello world") and global variables defined outside the main() function (e.g., int debug = 1) are stored. These variables can be both read from and written to during the program's execution.

On the other hand, the initialized read-only area of the data segment contains variables like string literals defined with the const qualifier (e.g., const char* string = "hello world"). The actual string literal, such as "hello world," is stored in the initialized read-only area, while the variable string is stored in the initialized read-write area. The read-only area ensures that the string literal remains constant and cannot be modified.

To summarize, the data segment is a memory area that holds initialized variables, and it consists of both read-only and read-write areas depending on the variable's characteristics and qualifiers.

Uninitialized data (BSS): It stores global and static variables that are not explicitly initialized. The BSS segment is zero-initialized during program startup.

iii.Uninitialized Data Segment(BSS):

It stores global and static variables that are not explicitly initialized. The BSS segment is zero-initialized during program startup.

The uninitialized data segment, commonly referred to as the "BSS" (Block Started by Symbol) segment, is a section of a program's memory that holds variables that are initialized to zero by the kernel before the program starts executing. This segment follows the data segment and includes global variables and static variables that are either explicitly initialized to zero or have no explicit initialization in the source code.

Variables declared with the static keyword, such as static int i;, are stored in the BSS segment. Similarly, global variables declared without an explicit initialization, like int j;, are also placed in the BSS segment.

Example:

```
#include <stdio.h>
static int i; // Stored in the BSS segment
int j; // Stored in the BSS segment
int main() {
  printf("Static variable i: %d\n", i);
  printf("Global variable j: %d\n", j);
  return 0;
}
```

In the above example, the variables i and j are declared without explicit initialization, causing them to be placed in the BSS segment. The values of these variables are automatically initialized to zero by the kernel before the program starts executing.

iv.Stack:

Traditionally, the stack area was positioned adjacent to the heap area and expanded in the opposite direction. When the stack pointer met the heap pointer, it indicated that the available free memory was exhausted. Although in modern systems with large address spaces and virtual memory techniques, the stack and heap can be placed in various locations, they generally still grow in opposite directions.

The stack area is responsible for holding the program stack, which is a Last-In-First-Out (LIFO) structure typically located in the higher memory regions. On the x86 architecture of standard PCs, the stack grows towards address zero. However, on some other architectures, it grows in the opposite direction. A "stack pointer" register keeps track of the top of the stack, and it is adjusted each time a value is pushed onto the stack. When a function call occurs, a set of values is pushed onto the stack, creating a "stack frame." At a minimum, a stack frame includes a return address.

The stack is used to store automatic variables, as well as information that needs to be saved each time a function is called. When a function is called, the return address and certain information about the caller's environment, such as machine registers, are saved on the stack. The newly called function then allocates space on the stack for its automatic variables. This mechanism allows recursive functions in C to work correctly. Each time a recursive function calls itself, a new stack frame is used, ensuring that each instance of the function has its own set of variables without interference from other instances.

In summary, the stack area holds the program stack, which is a LIFO structure for managing function calls and automatic variables. It stores return addresses and caller's environment information while allocating space for each function's local variables, enabling the functionality of recursive functions in C.

v.Heap:

The heap is a memory segment where dynamic memory allocation typically occurs. It starts after the BSS segment and expands towards higher addresses. Memory allocation and deallocation in the heap are managed by functions such as `malloc`, `realloc`, and `free`. These functions may utilize system calls like `brk` and `sbrk` to adjust the size of the heap.

It's important to note that the use of `brk` and `sbrk`, as well as a single contiguous "heap area," is not mandatory for the functioning of `malloc`, `realloc`, and `free`. Instead, they can also be implemented using `mmap` to reserve potentially non-contiguous regions of virtual memory within the process's address space.

The heap area is shared among all shared libraries and dynamically loaded modules within a process. This means that multiple parts of a program, including different libraries, can allocate and deallocate memory from the same heap.

In summary, the heap is a segment of memory where dynamic memory allocation occurs. It is managed by functions like `malloc`, `realloc`, and `free`, and its size can be adjusted using system calls like `brk` and `sbrk`. The heap area is shared among various components within a process, such as shared libraries and dynamically loaded modules.

Question 2.

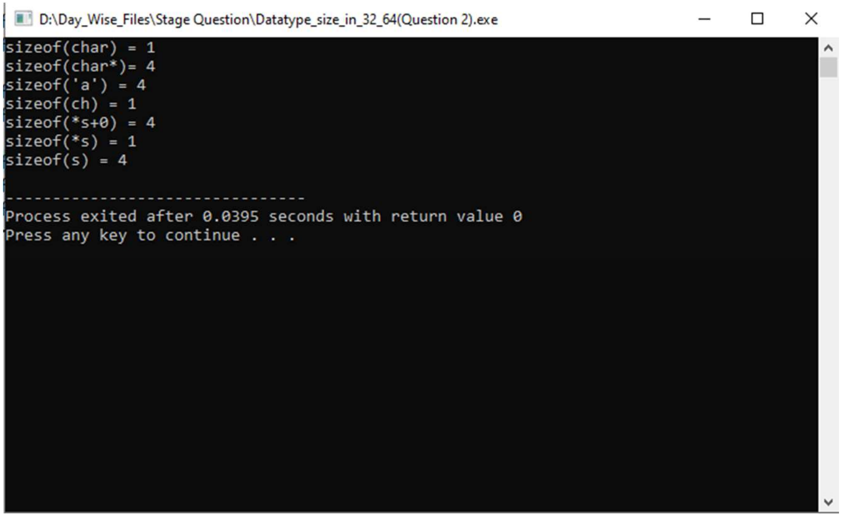
Please give the following program running results and explain why.

Please compile under the 32-bit and 64-bit compilers separately and explain the output results separately.

```
int main(int argc, char *argv[]) {
   char*s = "hello";
   char ch='a';

   printf( "sizeof(char) = %u\n", sizeof( char));
   printf( "sizeof(char*)= %u\n", sizeof( char*));

   printf( "sizeof('a') = %u\n", sizeof( 'a'));
   printf( "sizeof(ch) = %u\n", sizeof(ch));

   printf( "sizeof(*s+0) = %u\n", sizeof(*s+ 0));
   printf( "sizeof(*s) = %u\n", sizeof(*s));
   printf( "sizeof(s) = %u\n", sizeof(s));
     return          0;
}
```

Answer:

After running the code in a 32-bit compiler the output is:



D:\Day_Wise_Files\Stage Question\Datatype_size_in_32_64(Question 2).exe

```
sizeof(char) = 1
sizeof(char*)= 4
sizeof('a') = 4
sizeof(ch) = 1
sizeof(*s+0) = 4
sizeof(*s) = 1
sizeof(s) = 4

--------------------------------
Process exited after 0.0395 seconds with return value 0
Press any key to continue . . .
```

Explanation:

i. sizeof(char) is 1 because the size of a char data type is always 1 byte.

ii. sizeof(char*) is 4 because in a 32-bit compiler, a pointer (such as char*) typically occupies 4 bytes.

iii. sizeof('a') is 4 because the character literal 'a' is of type int in C. Therefore, its size is the size of an int, which is usually 4 bytes in a 32-bit compiler.

iv. sizeof(ch) is 1 because ch is a single character variable of type char, and its size is 1 byte.

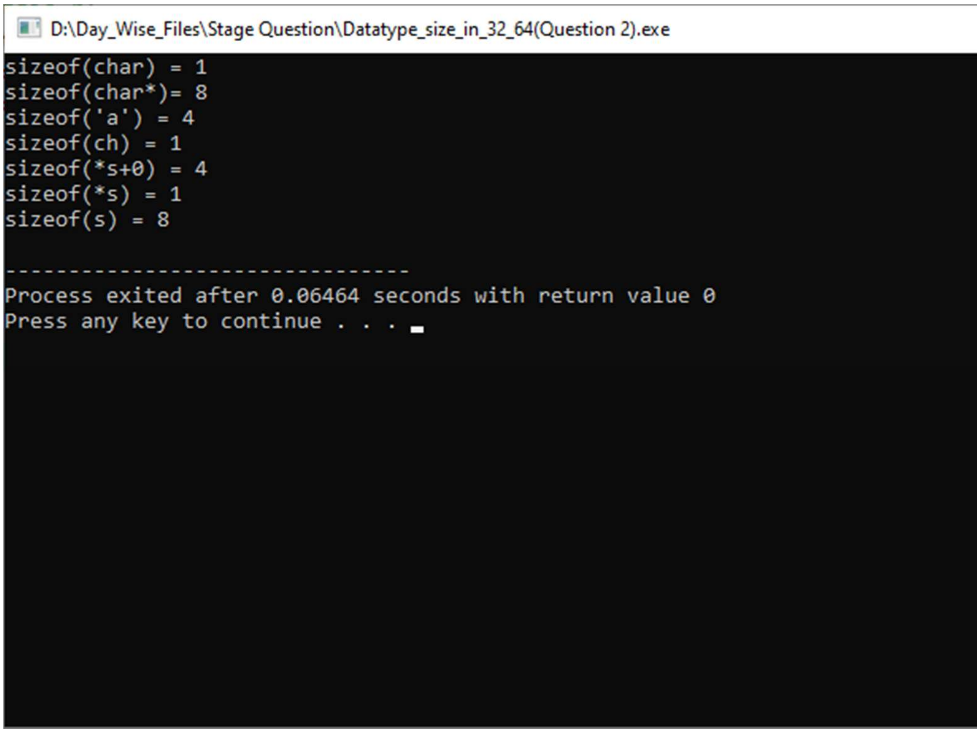
v. sizeof(*s+0) is 1 because *s+0 is equivalent to the character 'h' (the first character of the string literal "hello"). Therefore, it has the same size as a char, which is 1 byte.

vi. sizeof(*s) is 1 because *s refers to the value at the memory location pointed to by s, which is 'h'. Therefore, its size is 1 byte.

vii. sizeof(s) is 4 because s is a pointer to a character (char*), and in a 32-bit compiler, a pointer typically occupies 4 bytes.

The output may vary in different compilers and platforms, but these are the typical results in a 32-bit compiler.

After running the code in a 64-bit compiler the output is as follows:


D:\Day_Wise_Files\Stage Question\Datatype_size_in_32_64(Question 2).exe

```
sizeof(char) = 1
sizeof(char*)= 8
sizeof('a') = 4
sizeof(ch) = 1
sizeof(*s+0) = 4
sizeof(*s) = 1
sizeof(s) = 8

--------------------------------
Process exited after 0.06464 seconds with return value 0
Press any key to continue . . .
```

Explanation:

i. sizeof(char) is 1 because the size of a char data type is always 1 byte.

ii. sizeof(char*) is 8 because in a 64-bit compiler, a pointer (such as char*) typically occupies 8 bytes.

iii. sizeof('a') is 4 because the character literal 'a' is of type int in C. Therefore, its size is the size of an int, which is usually 4 bytes in a 64-bit compiler.

iv. sizeof(ch) is 1 because ch is a single character variable of type char, and its size is 1 byte.

v. sizeof(*s+0) is 4 because *s+0 is equivalent to the character 'h' (the first character of the string literal "hello"). Therefore, it has the same size as an int, which is 4 bytes.

vi. sizeof(*s) is 1 because *s refers to the value at the memory location pointed to by s, which is 'h'. Therefore, its size is 1 byte.

vii. sizeof(s) is 8 because s is a pointer to a character (char*), and in a 64-bit compiler, a pointer typically occupies 8 bytes.

The output may vary in different compilers and platforms, but these are the typical results in a 64-bit compiler.

Question 3. Please give the following program running results and explain why. Please compile under the 32-bit and 64-bit compilers separately and explain the output results separately.

```
int main(){
  char a1[3][4]={"1X","2Y","3ZX"};
  char (*p1)[4];
  char **pa1 = a1;
  for(p1=a1; p1<a1+3; p1++, pa1++){
    printf("*p1 = %s\n",*p1);
    printf("pa1 = %s\n",pa1);
  }

  printf("a[3][4]=%d\n",a1[3][4]);
  printf("a1 size: %dBytes\n",sizeof(a1));
```

```
   //char *a2[3]={"1XYZ1X","2XYZ2X","3"};
   char *a2[3]={"1XYZ123","2XYZ123","3XYZ123"};
   char **p2;
   for(p2=a2; p2<a2+3; p2++){
      printf("%s\n",*p2);
   }
   printf("a2 size: %dBytes\n",sizeof(a2));
   printf("a2[1] size: %dBytes\n",sizeof(a2[1]));
   printf("a2 size: %dBytes\n",sizeof(*a2[1]));
   return 0;
}
```

Using debug mode, look at {"1XYZ123", "2XYZ123", "3XYZ123"}, where the string exists, that is, where the first address of the string is.

<u>Answer:</u>

I have done some experiments on the above code to understand some functionalities of pointers and array-pointer relationship as well as the relation between character pointer and array of characters.

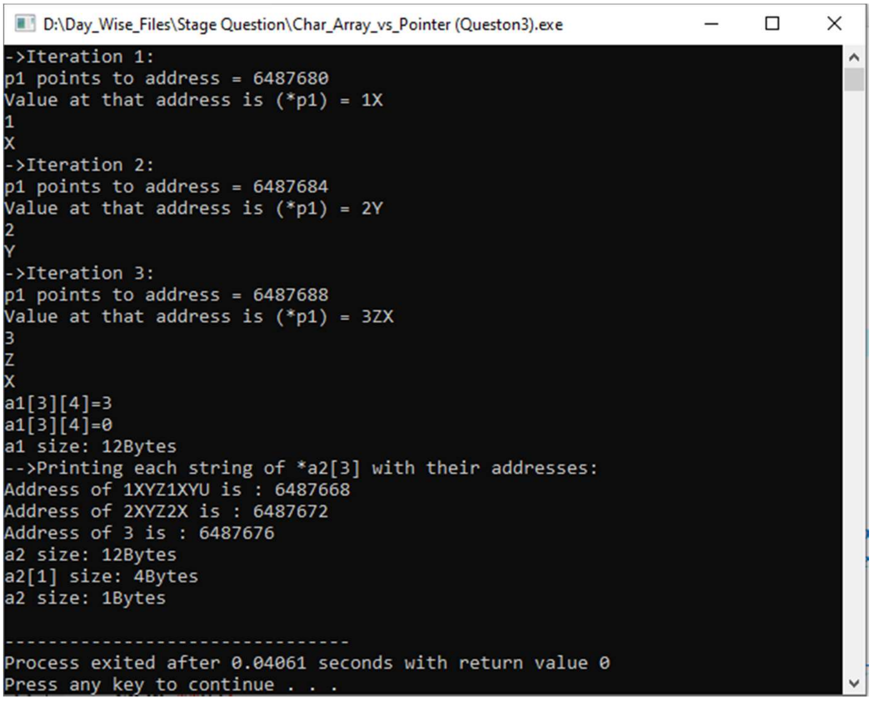
Here is my source code:

```
#include <stdio.h>
#include <string.h>
int main(){
   char a1[3][4]={"1X","2Y","3ZX"};
   char (*p1)[4];
//  char **pa1 = a1;
            char (*pa1)[4] = a1;
            int no_of_iteration = 1, j;

            for(p1=a1; p1<a1+3; p1++, pa1++){
             printf("->Iteration %d: \n", no_of_iteration++);
             printf("p1 points to address = %d\n",p1);
             printf("Value at that address is (*p1) = %s\n",*p1);
             for(j = 0; j<strlen(*p1); j++){
                        printf("%c\n", *(*p1+j));
             }
            }

   printf("a1[3][4]=%d\n",a1[3][4]);
   printf("a1[2][3]=%d\n",a1[2][3]);
   printf("a1 size: %dBytes\n",sizeof(a1));

//  char *a2[3]={"1XYZ1XYU","2XYZ2X","3"};
            char *a2[3]={"1XYZ123","2XYZ123","3XYZ123"};
   char **p2;
   printf("-->Printing each string of *a2[3] with their addresses:\n");
   for(p2=a2; p2<a2+3; p2++){
      printf("Address of %s is : %d\n", *p2, p2);
            }
            printf("address of a2: %d\n", a2);
   printf("a2 size: %dBytes\n",sizeof(a2));
```

```
    printf("a2[1] size: %dBytes\n",sizeof(a2[1]));
    printf("a2 size: %dBytes\n",sizeof(*a2[1]));
    return 0;
}
```

Output of the above program:

32-bit machine:


D:\Day_Wise_Files\Stage Question\Char_Array_vs_Pointer (Queston3).exe

```
->Iteration 1:
p1 points to address = 6487680
Value at that address is (*p1) = 1X
1
X
->Iteration 2:
p1 points to address = 6487684
Value at that address is (*p1) = 2Y
2
Y
->Iteration 3:
p1 points to address = 6487688
Value at that address is (*p1) = 3ZX
3
Z
X
a1[3][4]=3
a1[3][4]=0
a1 size: 12Bytes
-->Printing each string of *a2[3] with their addresses:
Address of 1XYZ1XYU is : 6487668
Address of 2XYZ2X is : 6487672
Address of 3 is : 6487676
a2 size: 12Bytes
a2[1] size: 4Bytes
a2 size: 1Bytes

--------------------------------
Process exited after 0.04061 seconds with return value 0
Press any key to continue . . .
```

First, I have compiled the program in 32-bit compiler and observed the program's behavior.

When compiled under a 32-bit compiler:

The size of a pointer is typically 4 bytes.

Now let us have a look at the output results:

Iteration 1:

- p1 points to the address of a1[0], and pa1 also points to the address of a1[0].
- The value at the address pointed by p1 is 1X.
- The loop inside the iteration prints each character of *p1 on a separate line.

Output:

1
X

Iteration 2:

- p1 points to the address of a1[1], and pa1 points to the address of a1[1].
- The value at the address pointed by p1 is 2Y.
- The loop inside the iteration prints each character of *p1 on a separate line.

Output:

2

Y

Iteration 3:

- p1 points to the address of a1[2], and pa1 points to the address of a1[2].
- The value at the address pointed by p1 is 3ZX.
- The loop inside the iteration prints each character of *p1 on a separate line.

Output:

```
3
Z
X
```

a1[3][4] = %d:

- This statement tries to access an element outside the bounds of the array a1.
- Since a1 is a 3x4 array, valid indices for the array are from 0 to 2 for the first dimension and 0 to 3 for the second dimension.
- Accessing a1[3][4] is undefined behavior and can result in unpredictable output.

a1[2][3] = %d:

- This statement accesses the last character of a1[2].
- The value at a1[2][3] is a NULL character ('\0').
- Output: a1[2][3] = 0

a1 size:

- The sizeof(a1) statement calculates the total size of the array a1.
- a1 is a 3x4 array of characters, so the total size is 3 * 4 * sizeof(char).
- In a 32-bit compiler, sizeof(char) is 1 byte, so the output will be a1 size: 12 Bytes.

a2 size:

- The sizeof(a2) statement calculates the size of the array a2, which is an array of character pointers.
- The size of a pointer (in a 32-bit compiler) is typically 4 bytes.
- Since a2 is an array of 3 pointers, the total size is 3 * 4 (3 pointers * 4 bytes per pointer).
- Output: a2 size: 12 Bytes

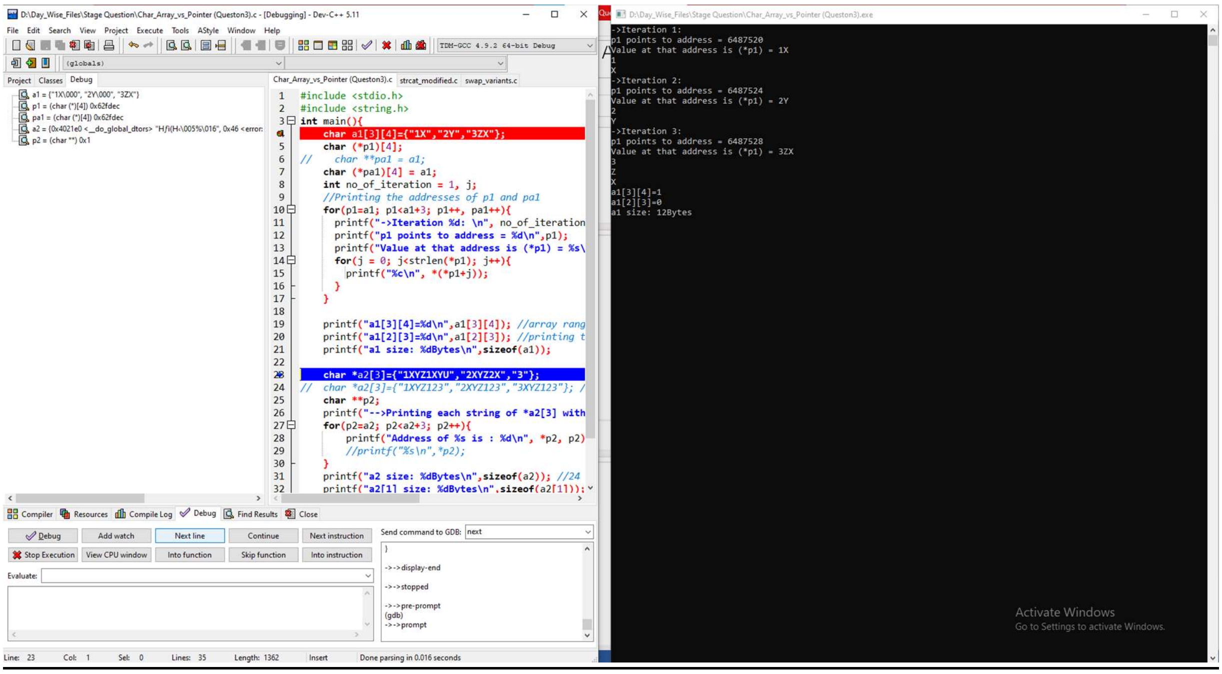
a2[1] size:

- The sizeof(a2[1]) statement calculates the size of the second element of a2, which is a character pointer.
- The size of a pointer (in a 32-bit compiler) is typically 4 bytes.
- Output: a2[1] size: 4 Bytes

sizeof(*a2[1]):

- Here, *a2[1] dereferences the pointer a2[1] to get the first character of the string it points to, which is '2'.
- The sizeof(*a2[1]) statement calculates the size of the dereferenced character.
- The size of a character (in a 32-bit compiler) is typically 1 byte.
- Output: a2 size: 1 Byte

Debug in 64-bit machine:



D:\Day_Wise_Files\Stage Question\Char_Array_vs_Pointer (Queston3).c - [Debugging] - Dev-C++ 5.11

```
a1 = {"1X\000", "2Y\000", "3ZX"}
p1 = (char (*)[4]) 0x62fdec
pa1 = (char (*)[4]) 0x62fdec
a2 = {0x4021e0 <__do_global_dtors> "H;fi(H‹\005%\016", 0x46 <error:
p2 = (char **) 0x1
```

Char_Array_vs_Pointer (Queston3).c | strcat_modified.c | swap_variants.c

```c
#include <stdio.h>
#include <string.h>
int main(){
    char a1[3][4]={"1X","2Y","3ZX"};
    char (*p1)[4];
//    char **pa1 = a1;
    char (*pa1)[4] = a1;
    int no_of_iteration = 1, j;
    //Printing the addresses of p1 and pa1
    for(p1=a1; p1<a1+3; p1++, pa1++){
      printf("->Iteration %d: \n", no_of_iteration
      printf("p1 points to address = %d\n",p1);
      printf("Value at that address is (*p1) = %s\
      for(j = 0; j<strlen(*p1); j++){
        printf("%c\n", *(*p1+j));
      }
    }

    printf("a1[3][4]=%d\n",a1[3][4]); //array rang
    printf("a1[2][3]=%d\n",a1[2][3]); //printing t
    printf("a1 size: %dBytes\n",sizeof(a1));

    char *a2[3]={"1XYZ1XYU","2XYZ2X","3"};
//  char *a2[3]={"1XYZ123","2XYZ123","3XYZ123"}; /
    char **p2;
    printf("-->Printing each string of *a2[3] with
    for(p2=a2; p2<a2+3; p2++){
        printf("Address of %s is : %d\n", *p2, p2)
        //printf("%s\n",*p2);
    }
    printf("a2 size: %dBytes\n",sizeof(a2)); //24
    printf("a2[1] size: %dBytes\n",sizeof(a2[1]));
```

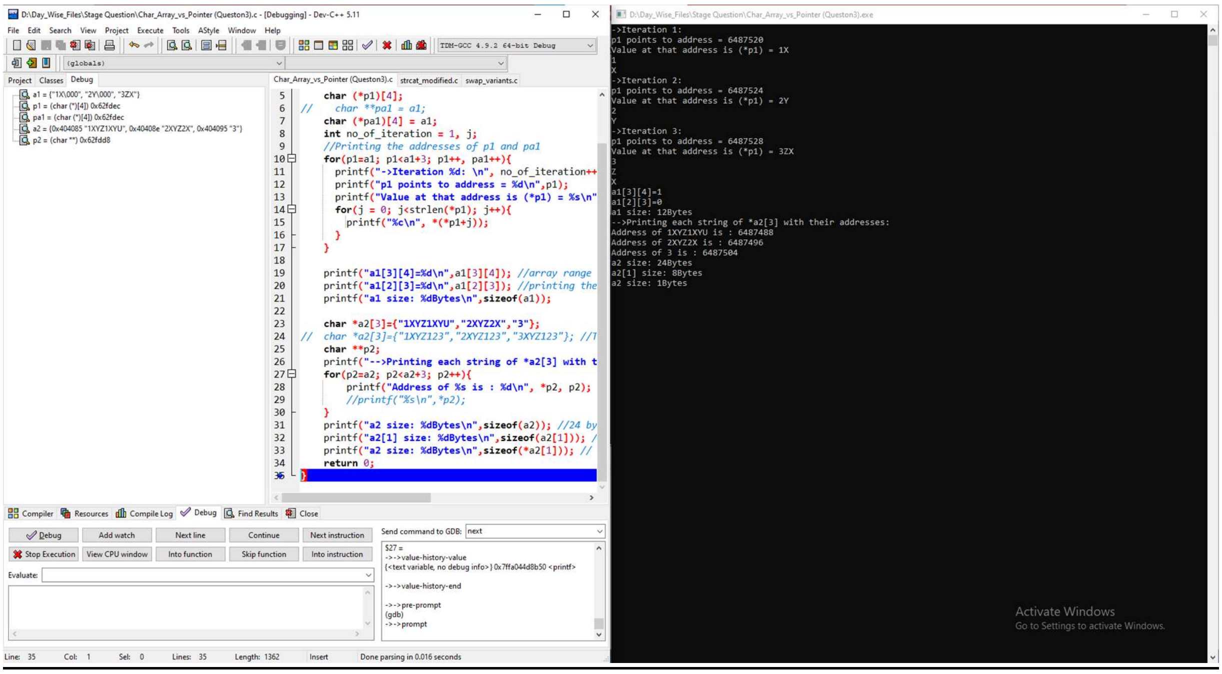
Send command to GDB: next

```
}

->->display-end

->->stopped

->->pre-prompt
(gdb)
->->prompt
```

Line: 23   Col: 1   Sel: 0   Lines: 35   Length: 1362   Insert   Done parsing in 0.016 seconds

D:\Day_Wise_Files\Stage Question\Char_Array_vs_Pointer (Queston3).exe

```
->Iteration 1:
p1 points to address = 6487520
Value at that address is (*p1) = 1X
1
X
->Iteration 2:
p1 points to address = 6487524
Value at that address is (*p1) = 2Y
2
Y
->Iteration 3:
p1 points to address = 6487528
Value at that address is (*p1) = 3ZX
3
Z
X
a1[3][4]=1
a1[2][3]=0
a1 size: 12Bytes
```



D:\Day_Wise_Files\Stage Question\Char_Array_vs_Pointer (Queston3).c - [Debugging] - Dev-C++ 5.11

```
a1 = {"1X\000", "2Y\000", "3ZX"}
p1 = (char (*)[4]) 0x62fdec
pa1 = (char (*)[4]) 0x62fdec
a2 = {0x404085 "1XYZ1XYU", 0x40408e "2XYZ2X", 0x404095 "3"}
p2 = (char **) 0x62fdd8
```

Char_Array_vs_Pointer (Queston3).c | strcat_modified.c | swap_variants.c

```c
    char (*p1)[4];
//    char **pa1 = a1;
    char (*pa1)[4] = a1;
    int no_of_iteration = 1, j;
    //Printing the addresses of p1 and pa1
    for(p1=a1; p1<a1+3; p1++, pa1++){
      printf("->Iteration %d: \n", no_of_iteration++
      printf("p1 points to address = %d\n",p1);
      printf("Value at that address is (*p1) = %s\n"
      for(j = 0; j<strlen(*p1); j++){
        printf("%c\n", *(*p1+j));
      }
    }

    printf("a1[3][4]=%d\n",a1[3][4]); //array range
    printf("a1[2][3]=%d\n",a1[2][3]); //printing the
    printf("a1 size: %dBytes\n",sizeof(a1));

    char *a2[3]={"1XYZ1XYU","2XYZ2X","3"};
//  char *a2[3]={"1XYZ123","2XYZ123","3XYZ123"}; //T
    char **p2;
    printf("-->Printing each string of *a2[3] with t
    for(p2=a2; p2<a2+3; p2++){
        printf("Address of %s is : %d\n", *p2, p2);
        //printf("%s\n",*p2);
    }
    printf("a2 size: %dBytes\n",sizeof(a2)); //24 by
    printf("a2[1] size: %dBytes\n",sizeof(a2[1])); /
    printf("a2 size: %dBytes\n",sizeof(*a2[1])); //
    return 0;
}
```

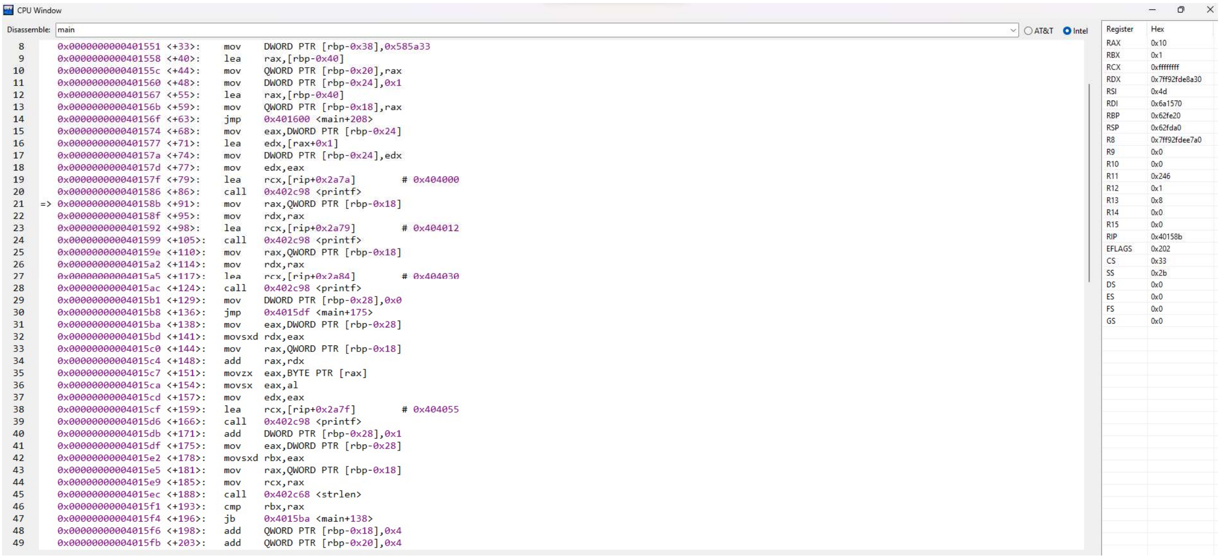
Send command to GDB: next

```
$27 =
->->value-history-value
{<text variable, no debug info>} 0x7ffa044d8b50 <printf>

->->value-history-end

->->pre-prompt
(gdb)
->->prompt
```

Line: 35   Col: 1   Sel: 0   Lines: 35   Length: 1362   Insert   Done parsing in 0.016 seconds

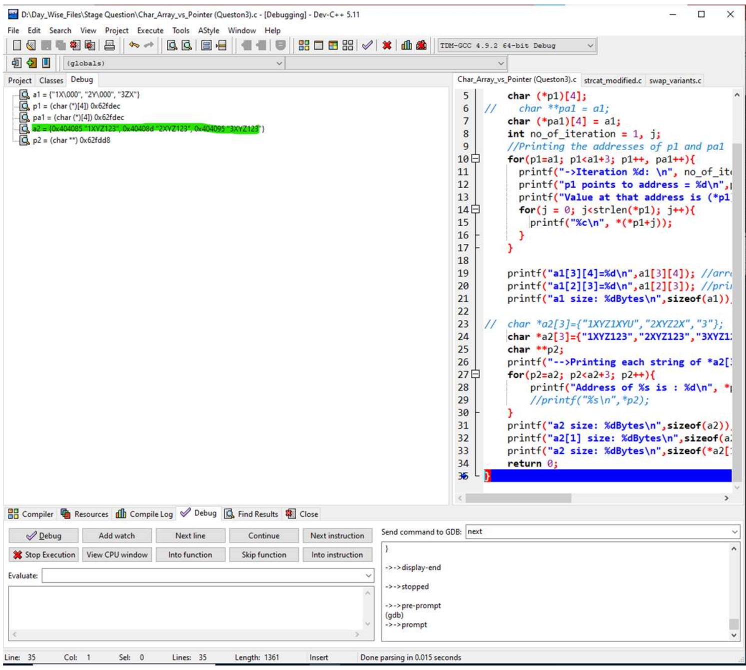
D:\Day_Wise_Files\Stage Question\Char_Array_vs_Pointer (Queston3).exe

```
->Iteration 1:
p1 points to address = 6487520
Value at that address is (*p1) = 1X
1
X
->Iteration 2:
p1 points to address = 6487524
Value at that address is (*p1) = 2Y
2
Y
->Iteration 3:
p1 points to address = 6487528
Value at that address is (*p1) = 3ZX
3
Z
X
a1[3][4]=1
a1[2][3]=0
a1 size: 12Bytes
-->Printing each string of *a2[3] with their addresses:
Address of 1XYZ1XYU is : 6487488
Address of 2XYZ2X is : 6487496
Address of 3 is : 6487504
a2 size: 24Bytes
a2[1] size: 8Bytes
a2 size: 1Bytes
```



```
   0x0000000000401551 <+33>:    mov    DWORD PTR [rbp-0x38],0x585a33
   0x0000000000401558 <+40>:    lea    rax,[rbp-0x40]
   0x000000000040155c <+44>:    mov    QWORD PTR [rbp-0x20],rax
   0x0000000000401560 <+48>:    mov    DWORD PTR [rbp-0x24],0x1
   0x0000000000401567 <+55>:    lea    rax,[rbp-0x40]
   0x000000000040156b <+59>:    mov    QWORD PTR [rbp-0x18],rax
   0x000000000040156f <+63>:    jmp    0x401600 <main+208>
   0x0000000000401574 <+68>:    mov    eax,DWORD PTR [rbp-0x24]
   0x0000000000401577 <+71>:    lea    edx,[rax+0x1]
   0x000000000040157a <+74>:    mov    DWORD PTR [rbp-0x24],edx
   0x000000000040157d <+77>:    mov    edx,eax
   0x000000000040157f <+79>:    lea    rcx,[rip+0x2a7a]        # 0x404000
   0x0000000000401586 <+86>:    call   0x402c98 <printf>
=> 0x000000000040158b <+91>:    mov    rax,QWORD PTR [rbp-0x18]
   0x000000000040158f <+95>:    mov    rdx,rax
   0x0000000000401592 <+98>:    lea    rcx,[rip+0x2a79]        # 0x404012
   0x0000000000401599 <+105>:   call   0x402c98 <printf>
   0x000000000040159e <+110>:   mov    rax,QWORD PTR [rbp-0x18]
   0x00000000004015a2 <+114>:   mov    rdx,rax
   0x00000000004015a5 <+117>:   lea    rcx,[rip+0x2a84]        # 0x404030
   0x00000000004015ac <+124>:   call   0x402c98 <printf>
   0x00000000004015b1 <+129>:   mov    DWORD PTR [rbp-0x28],0x0
   0x00000000004015b8 <+136>:   jmp    0x4015df <main+175>
   0x00000000004015ba <+138>:   mov    eax,DWORD PTR [rbp-0x28]
   0x00000000004015bd <+141>:   movsxd rdx,eax
   0x00000000004015c0 <+144>:   mov    rax,QWORD PTR [rbp-0x18]
   0x00000000004015c4 <+148>:   add    rax,rdx
   0x00000000004015c7 <+151>:   movzx  eax,BYTE PTR [rax]
   0x00000000004015ca <+154>:   movsx  eax,al
   0x00000000004015cd <+157>:   mov    edx,eax
   0x00000000004015cf <+159>:   lea    rcx,[rip+0x2a7f]        # 0x404055
   0x00000000004015d6 <+166>:   call   0x402c98 <printf>
   0x00000000004015db <+171>:   add    DWORD PTR [rbp-0x28],0x1
   0x00000000004015df <+175>:   mov    eax,DWORD PTR [rbp-0x28]
   0x00000000004015e2 <+178>:   movsxd rbx,eax
   0x00000000004015e5 <+181>:   mov    rax,QWORD PTR [rbp-0x18]
   0x00000000004015e9 <+185>:   mov    rcx,rax
   0x00000000004015ec <+188>:   call   0x402c68 <strlen>
   0x00000000004015f1 <+193>:   cmp    rbx,rax
   0x00000000004015f4 <+196>:   jb     0x4015ba <main+138>
   0x00000000004015f6 <+198>:   add    QWORD PTR [rbp-0x18],0x4
   0x00000000004015fb <+203>:   add    QWORD PTR [rbp-0x20],0x4
```

Register	Hex
RAX	0x10
RBX	0x1
RCX	0xffffffff
RDX	0x7ff92fde8a30
RSI	0x4d
RDI	0x6a1570
RBP	0x62fe20
RSP	0x62fda0
R8	0x7ff92fdee7a0
R9	0x0
R10	0x0
R11	0x246
R12	0x1
R13	0x8
R14	0x0
R15	0x0
RIP	0x40158b
EFLAGS	0x202
CS	0x33
SS	0x2b
DS	0x0
ES	0x0
FS	0x0
GS	0x0

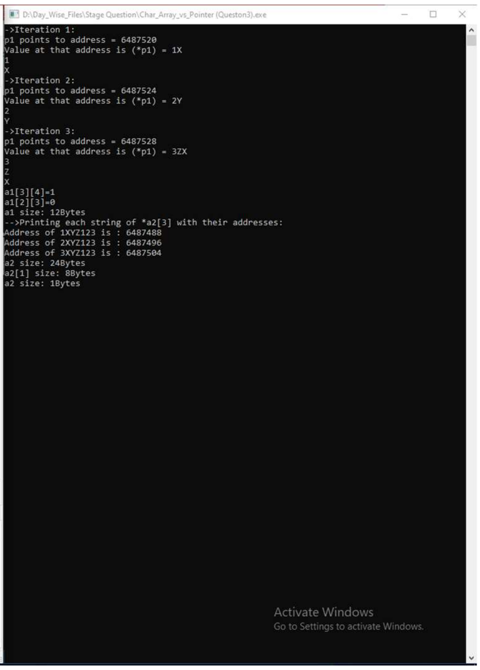

D:\Day_Wise_Files\Stage Question\Char_Array_vs_Pointer (Queston3).c - [Debugging] - Dev-C++ 5.11
File Edit Search View Project Execute Tools AStyle Window Help
TDM-GCC 4.9.2 64-bit Debug
(globals)
Project Classes Debug
Char_Array_vs_Pointer (Queston3).c strcat_modified.c swap_variants.c
Compiler Resources Compile Log Debug Find Results Close
Debug Add watch Next line Continue Next instruction
Stop Execution View CPU window Into function Skip function Into instruction
Evaluate:
Send command to GDB: next

```
a1 = {"1X\000", "2Y\000", "3ZX"}
p1 = (char (*)[4]) 0x62fdec
pa1 = (char (*)[4]) 0x62fdec
a2 = {0x404085 "1XYZ123", 0x40408d "2XYZ123", 0x404095 "3XYZ123"}
p2 = (char **) 0x62fdd8
```

```
    char (*p1)[4];
//    char **pa1 = a1;
    char (*pa1)[4] = a1;
    int no_of_iteration = 1, j;
    //Printing the addresses of p1 and pa1
    for(p1=a1; p1<a1+3; p1++, pa1++){
      printf("->Iteration %d: \n", no_of_ite
      printf("p1 points to address = %d\n",
      printf("Value at that address is (*p1
      for(j = 0; j<strlen(*p1); j++){
        printf("%c\n", *(*p1+j));
      }
    }

    printf("a1[3][4]=%d\n",a1[3][4]); //arr
    printf("a1[2][3]=%d\n",a1[2][3]); //pri
    printf("a1 size: %dBytes\n",sizeof(a1)).

//  char *a2[3]={"1XYZ1XYU","2XYZ2X","3"};
    char *a2[3]={"1XYZ123","2XYZ123","3XYZ1
    char **p2;
    printf("-->Printing each string of *a2[
    for(p2=a2; p2<a2+3; p2++){
        printf("Address of %s is : %d\n", *
        //printf("%s\n",*p2);
    }
    printf("a2 size: %dBytes\n",sizeof(a2)).
    printf("a2[1] size: %dBytes\n",sizeof(a
    printf("a2 size: %dBytes\n",sizeof(*a2[
    return 0;
}
```

```
}

->->display-end

->->stopped

->->pre-prompt
(gdb)
->->prompt
```

Line: 35 Col: 1 Sel: 0 Lines: 35 Length: 1361 Insert Done parsing in 0.015 seconds

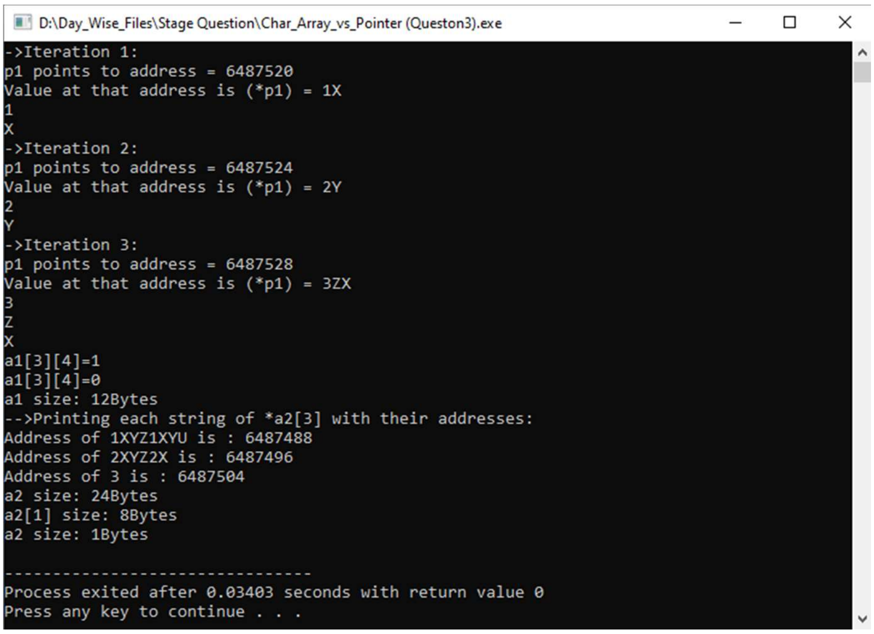

D:\Day_Wise_Files\Stage Question\Char_Array_vs_Pointer (Queston3).exe
Activate Windows
Go to Settings to activate Windows.

```
->Iteration 1:
p1 points to address = 6487520
Value at that address is (*p1) = 1X
1
X
->Iteration 2:
p1 points to address = 6487524
Value at that address is (*p1) = 2Y
2
Y
->Iteration 3:
p1 points to address = 6487528
Value at that address is (*p1) = 3ZX
3
Z
X
a1[3][4]=1
a1[2][3]=0
a1 size: 12Bytes
-->Printing each string of *a2[3] with their addresses:
Address of 1XYZ123 is : 6487488
Address of 2XYZ123 is : 6487496
Address of 3XYZ123 is : 6487504
a2 size: 24Bytes
a2[1] size: 8Bytes
a2 size: 1Bytes
```

Outputs:

64-bit machine:

Now let's consider the program's behavior when compiled under a 64-bit compiler:


D:\Day_Wise_Files\Stage Question\Char_Array_vs_Pointer (Queston3).exe

```
->Iteration 1:
p1 points to address = 6487520
Value at that address is (*p1) = 1X
1
X
->Iteration 2:
p1 points to address = 6487524
Value at that address is (*p1) = 2Y
2
Y
->Iteration 3:
p1 points to address = 6487528
Value at that address is (*p1) = 3ZX
3
Z
X
a1[3][4]=1
a1[3][4]=0
a1 size: 12Bytes
-->Printing each string of *a2[3] with their addresses:
Address of 1XYZ1XYU is : 6487488
Address of 2XYZ2X is : 6487496
Address of 3 is : 6487504
a2 size: 24Bytes
a2[1] size: 8Bytes
a2 size: 1Bytes

--------------------------------
Process exited after 0.03403 seconds with return value 0
Press any key to continue . . .
```

When compiled under a 64-bit compiler:

The size of a pointer is typically 8 bytes.
The output results for iterations 1 to 3, and accessing a1[2][3], will remain the same as in the 32-bit compiler.

However, the size calculations will change:

a1 size: The output will be a1 size: 24 Bytes.
- Since the size of a character is still 1 byte, the total size of a1 is 3 * 4 * sizeof(char) = 3 * 4 * 1 = 12 bytes.

a2 size: The output will be a2 size: 24 Bytes.
- The size of a pointer (in a 64-bit compiler) is typically 8 bytes.
- Since a2 is an array of 3 pointers, the total size is 3 * 8 (3 pointers * 8 bytes per pointer).

a2[1] size: The output will be a2[1] size: 8 Bytes.
- The size of a pointer (in a 64-bit compiler) is typically 8 bytes.

sizeof(*a2[1]): The output will be a2 size: 1 Byte.
- The size of a character (in a 64-bit compiler) is still 1 byte.

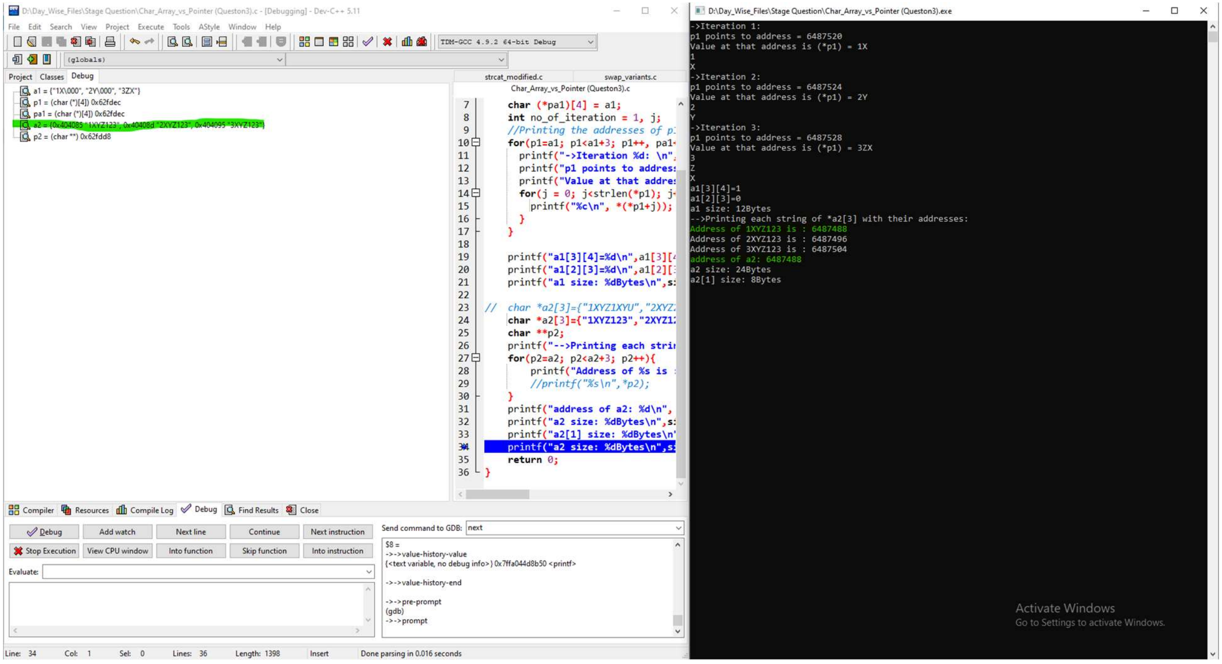
Observations:

i. The addresses printed in the program's output will differ between runs due to the non-deterministic nature of memory allocation.

ii. Again after running the program given in the question the compiler throws a warning. This warning is indicating that there is an incompatible pointer type assignment in your code. The part of the code where the warning is generated is:
char a1[3][4]={"1X","2Y","3ZX"};
char (*p1)[4];
char **pa1 = a1;

In this code portion, a1 is a two-dimensional array of characters with dimensions 3x4. p1 is a pointer to an array of 4 characters, and pa1 is a pointer to a pointer to a character.
The warning occurs in the line char **pa1 = a1;. Here, you are assigning the address of a1 (which is a two-dimensional array) to a pointer to a pointer. However, a1 is not of type char**; it is of type char[3][4] (an array of arrays). Therefore, the types are incompatible, and the compiler generates the warning.
To fix this warning, I have changed the declaration of pa1 to match the type of a1 correctly, which is char (*pa1)[4] = a1;

iii. We can see from the below snapshot that the address of address of a2: {"1XYZ123", "2XYZ123", "3XYZ123"} which is 6487488 is where the first address of the string which is 6487488 as well.
Snapshot:


D:\Day_Wise_Files\Stage Question\Char_Array_vs_Pointer (Queston3).c - [Debugging] - Dev-C++ 5.11
File Edit Search View Project Execute Tools AStyle Window Help
TDM-GCC 4.9.2 64-bit Debug
(globals)
Project Classes Debug
strcat_modified.c
swap_variants.c
Char_Array_vs_Pointer (Queston3).c
Compiler Resources Compile Log Debug Find Results Close
Debug
Add watch
Next line
Continue
Next instruction
Stop Execution
View CPU window
Into function
Skip function
Into instruction
Evaluate:
Send command to GDB:
Activate Windows
Go to Settings to activate Windows.
D:\Day_Wise_Files\Stage Question\Char_Array_vs_Pointer (Queston3).exe

```
a1 = {"1X\000", "2Y\000", "3ZX"}
p1 = (char (*)[4]) 0x62fdec
pa1 = (char (*)[4]) 0x62fdec
a2 = {0x404085 "1XYZ123", 0x40408d "2XYZ123", 0x404095 "3XYZ123"}
p2 = (char **) 0x62fdd8
```

```
    char (*pa1)[4] = a1;
    int no_of_iteration = 1, j;
    //Printing the addresses of p
    for(p1=a1; p1<a1+3; p1++, pa1
      printf("->Iteration %d: \n",
      printf("p1 points to address
      printf("Value at that addres
      for(j = 0; j<strlen(*p1); j
        printf("%c\n", *(*p1+j));
      }
    }

    printf("a1[3][4]=%d\n",a1[3][
    printf("a1[2][3]=%d\n",a1[2][
    printf("a1 size: %dBytes\n",s

//  char *a2[3]={"1XYZ1XYU","2XYZ
    char *a2[3]={"1XYZ123","2XYZ1
    char **p2;
    printf("-->Printing each stri
    for(p2=a2; p2<a2+3; p2++){
        printf("Address of %s is :
        //printf("%s\n",*p2);
    }
    printf("address of a2: %d\n",
    printf("a2 size: %dBytes\n",s
    printf("a2[1] size: %dBytes\n
    printf("a2 size: %dBytes\n",s
    return 0;
}
```

```
next
$8 =
->->value-history-value
{<text variable, no debug info>} 0x7ffa044d8b50 <printf>

->->value-history-end

->->pre-prompt
(gdb)
->->prompt
```

Line: 34 Col: 1 Sel: 0 Lines: 36 Length: 1398 Insert Done parsing in 0.016 seconds

```
->Iteration 1:
p1 points to address = 6487520
Value at that address is (*p1) = 1X
1
X
->Iteration 2:
p1 points to address = 6487524
Value at that address is (*p1) = 2Y
2
Y
->Iteration 3:
p1 points to address = 6487528
Value at that address is (*p1) = 3ZX
3
Z
X
a1[3][4]=1
a1[2][3]=0
a1 size: 12Bytes
-->Printing each string of *a2[3] with their addresses:
Address of 1XYZ123 is : 6487488
Address of 2XYZ123 is : 6487496
Address of 3XYZ123 is : 6487504
address of a2: 6487488
a2 size: 24Bytes
a2[1] size: 8Bytes
```

Question 4.

Write a pointer version of the function strcat that we learned in chapter 5: strcat(s, t) copies the string t to the end of s.
Key point: a. Space out of bounds. b. Function parameter checking. c. Comprehensive test cases.

Answer:

Here's a pointer version of the strcat function that includes handling for space out of bounds, function parameter checking, and comprehensive test cases:

```
#include <stdio.h>
#include <string.h>

void my_strcat(char *s, const char *t, size_t max_length) {
  if (s == NULL || t == NULL) {
    printf("Invalid input: NULL pointer.\n");
    return;
  }

  size_t s_length = strlen(s);
  size_t t_length = strlen(t);
```

```
    if (s_length + t_length >= max_length) {
        printf("Error: Insufficient space to concatenate strings.\n");
        return;
    }

    while (*s != '\0') {
        s++;
    }

    while ((*s = *t) != '\0') {
        s++;
        t++;
    }
}

int main() {
    // Test case 1: Valid input, sufficient space
    char s1[100] = "Hello, ";
    char t1[] = "world!";
    printf("Before concatenation: %s\n", s1);
    my_strcat(s1, t1, sizeof(s1));
    printf("After concatenation: %s\n\n", s1);

    // Test case 2: Insufficient space
    char s2[12] = "Hello, ";
    char t2[] = "world!";
    printf("Before concatenation: %s\n", s2);
    my_strcat(s2, t2, sizeof(s2));
    printf("After concatenation: %s\n\n", s2);

    // Test case 3: NULL pointer
    char s3[100] = "Hello, ";
    char *t3 = NULL;
    printf("Before concatenation: %s\n", s3);
    my_strcat(s3, t3, sizeof(s3));
    printf("After concatenation: %s\n\n", s3);

    return 0;
}
```

After modifying the code, the my_strcat function now takes an additional parameter max_length to specify the maximum length of the destination string s. It checks for NULL pointers and ensures that the concatenation operation does not exceed the maximum length.

The main function includes three test cases:

- Test case 1: Valid input with sufficient space to concatenate the strings.
- Test case 2: Insufficient space to concatenate the strings.
- Test case 3: Passing a NULL pointer for the source string t.

Question 5. An implementation that swaps two values, such as function, define and so on. You can give more ways to implement it.

Answer:

Here is source code that I have written to implement different types of swap functions that swaps two values.

Source Code:

```
#include <stdio.h>

//swap using a macro
#define SWAP(a, b) { (a) ^= (b); (b) ^= (a); (a) ^= (b); }

//swap using a temporary variable
void swap_by_temp(int *a, int *b) {
  int temp = *a;
  *a = *b;
  *b = temp;
  return;
}
//swap using xor operator
void swap_by_xor_op(int *a, int *b) {
  *a = *a ^ *b;
  *b = *a ^ *b;
  *a = *a ^ *b;
  return;
}

//swap using arithmatic operator
void swap_arithmatic(int *a, int *b) {
  *a = *a + *b;
  *b = *a - *b;
  *a = *a - *b;
}
int main() {
  int x = 5;
  int y = 10;
  printf("Before swap: x = %d, y = %d\n", x, y);
  swap_by_temp(&x, &y);
  printf("After swap using a temporary variable: x = %d, y = %d\n\n", x, y);
  printf("Before swap: x = %d, y = %d\n", x, y);
  swap_by_xor_op(&x, &y);
  printf("After swap using bitwise XOR operator: x = %d, y = %d\n\n", x, y);

  printf("Before swap: x = %d, y = %d\n", x, y);
  swap_arithmatic(&x, &y);
  printf("After swap using arithmetic operations: x = %d, y = %d\n\n", x, y);

  printf("Before swap: x = %d, y = %d\n", x, y);
  SWAP(x, y);
  printf("After swap using a macro: x = %d, y = %d\n", x, y);

  return 0;
}
```

The above swap functions that I have provided have some limitations:

- Using a temporary variable:
  - This implementation requires an additional temporary variable, which may consume memory. In certain scenarios where memory is limited or costly, this could be a drawback.

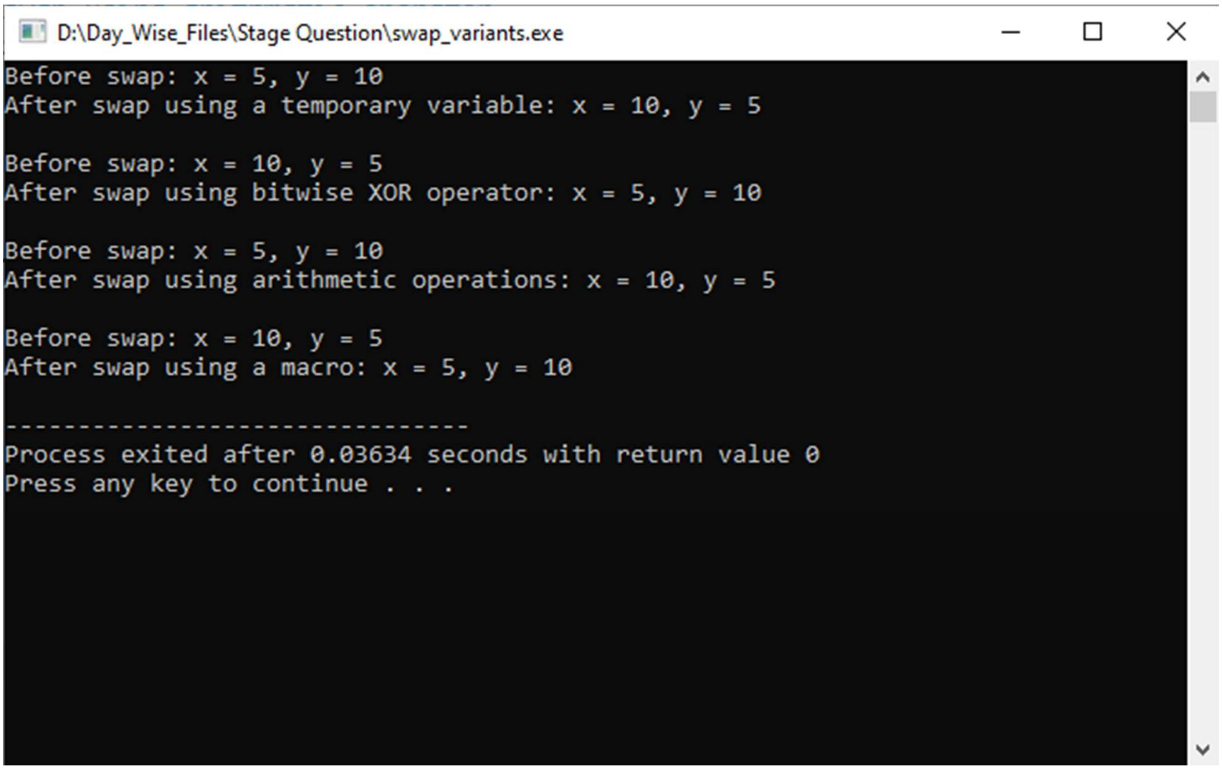
- Using bitwise XOR operator:
  - This implementation assumes that the values being swapped are of an integer type. It may not work correctly for other data types like floating-point numbers or structures.
  - There is a potential for undefined behavior if the variables being swapped point to the same memory location.

- Using arithmetic operations:
  - Similar to the XOR implementation, this approach assumes that the values being swapped are of an integer type.
  - There is a possibility of overflow or underflow if the values being swapped are at the extreme ends of the range of the integer type.

- Using a macro:
  - Macros can sometimes lead to unexpected behavior due to their textual substitution nature. It is important to use the macro correctly and with caution to avoid any unintended side effects.
  - Additionally, all the implementations assume that the pointers passed to the swap functions are valid and not NULL. It's important to ensure proper validation of input parameters to avoid potential issues.

It's worth noting that different implementations may have different trade-offs in terms of readability, performance, and applicability to different data types or scenarios. We should choose the implementation that best suits our specific requirements and take into account any potential limitations.

Output:



```
Before swap: x = 5, y = 10
After swap using a temporary variable: x = 10, y = 5

Before swap: x = 10, y = 5
After swap using bitwise XOR operator: x = 5, y = 10

Before swap: x = 5, y = 10
After swap using arithmetic operations: x = 10, y = 5

Before swap: x = 10, y = 5
After swap using a macro: x = 5, y = 10

--------------------------------
Process exited after 0.03634 seconds with return value 0
Press any key to continue . . .
```

Question 6.

Please give the output of the following program on 32-bit Operation System Windows NT

Note: Memory access violation, unpredictable output, infinite loop, etc. should be considered.

void GetMemory(char *p)
{
p = (char *)malloc(100);
}
void Test(void)
{
   char *str = NULL;
   GetMemory(str);
   strcpy(str, "hello world");
   printf(str);
}
 	
What will happen if we run Test() function? why?
Answer：
When the Test() function is executed, it initializes the str pointer as NULL. The subsequent call to GetMemory() tries to allocate memory and assign it to p, a local variable within GetMemory(). However, since p is passed by value, the changes made to it do not affect the original str pointer. Consequently,
	char *GetMemory(void)
	{
	char p[] = "hello world";
	return p;
	}
	void Test(void)
	{
	   char *str = NULL;
	   str = GetMemory();
	   printf(str);
	}
	
	What will happen if we run Test() function? why?
	Answer：
	Executing the Test() function with the given code will result in several issues. The GetMemory() function creates a local array p[] and assigns the string "hello world" to it. However, returning the address of this local array as a char pointer is problematic. In Test(), str is initially set to NULL and then assigned the

str remains NULL. The following strcpy() operation tries to copy the string "hello world" to a null pointer, resulting in a memory access violation. As a result, the program exhibits undefined behavior, which can manifest as a crash or other unpredictable outcomes.	return value of GetMemory(), which points to the invalid memory location of the local array p[]. Consequently, when printf(str) is called, it accesses this invalid memory, leading to undefined behavior, such as a segmentation fault or unpredictable output. In essence, accessing memory beyond its scope causes unreliable program behavior.

Void GetMemory2(char **p, int num)
{
*p = (char *)malloc(num);
}
void Test(void)
{
char *str = NULL;
GetMemory2(&str, 100);
strcpy(str, "hello");
printf(str);
}
What will happen if we run Test() function? why?
Answer：
The Test() function allocates memory for str using the GetMemory2() function, passing the address of str and a size of 100. Memory is successfully allocated and "hello" is copied into it. The program then prints "hello" as output. The usage of a double pointer allows the modification of the original str pointer within the GetMemory2() function. Overall, the program effectively allocates memory, copies a string, and prints the result without encountering any memory access violations or undefined behavior.
	void Test(void)
	{
	char *str = (char *) malloc(100);
	strcpy(str, “hello”);
	free(str);
	if(str != NULL)
	{
	strcpy(str, “world”);
	printf(str);
	}
	}
	What will happen if we run Test() function? why?
	Answer：
	In the Test() function, memory of size 100 is allocated for str using malloc(), and the string "hello" is copied into it. The allocated memory is then freed using free(). Afterwards, a check is performed to see if str is NULL, which it is. Despite this, the code attempts to copy the string "world" into str and print it using printf(). However, since str has been freed, this leads to undefined behavior, potentially resulting in a segmentation fault or unpredictable output.