1. Take an example C program in which main function calls any other function with 3 or more call by value parameters. Find out how and when values of actual parameters are passed to the formal parameters in the called function. Also point out when and where main function (or any other function) copies return address to the called function. Note: refer chapter 9 and 10 of the book

We will use the following C program. This program initializes four integer variables inside the main function and calls another function called sum which adds all four variables that are passed to it as parameters and returns their sum.

sum.c file :

```
int sum(int a, int b, int c, int d) {
        return a + b + c + d;
}

int main() {
        int x = 2;
        int y = 3;
        int z = 5;
        int w = 7;
        int r = sum(x, y, z, w);
        return 0;
}
```

To see its equivalent assembly code and debug using gdb, we will generate the object file with the following command.

```
gcc -g sum.c -O0 -o sum
```

We can get the assembly code of the main & sum function using either “gdb disassemble” command or “objdump -d sum” command. I’ve used the latter option as it gives the assembly code of the entire program at once rather than calling “gdb disassemble” at breakpoints of main and sum functions.

Assembly code for the main function is as follows.

```
000000000040112a <main>:
  40112a:   55                      push   %rbp
  40112b:   48 89 e5                mov    %rsp,%rbp
  40112e:   48 83 ec 20             sub    $0x20,%rsp
  401132:   c7 45 fc 02 00 00 00    movl   $0x2,-0x4(%rbp)
  401139:   c7 45 f8 03 00 00 00    movl   $0x3,-0x8(%rbp)
  401140:   c7 45 f4 05 00 00 00    movl   $0x5,-0xc(%rbp)
```

```
401147:    c7 45 f0 07 00 00 00   movl  $0x7,-0x10(%rbp)
40114e:    8b 4d f0               mov   -0x10(%rbp),%ecx
401151:    8b 55 f4               mov   -0xc(%rbp),%edx
401154:    8b 75 f8               mov   -0x8(%rbp),%esi
401157:    8b 45 fc               mov   -0x4(%rbp),%eax
40115a:    89 c7                  mov   %eax,%edi
40115c:    e8 a5 ff ff ff         callq 401106 <sum>
401161:    89 45 ec               mov   %eax,-0x14(%rbp)
401164:    b8 00 00 00 00         mov   $0x0,%eax
401169:    c9                     leaveq
40116a:    c3                     retq
40116b:    0f 1f 44 00 00         nopl  0x0(%rax,%rax,1)
```

Assembly code for the sum function is as follows.

```
0000000000401106 <sum>:
401106:    55                     push  %rbp
401107:    48 89 e5               mov   %rsp,%rbp
40110a:    89 7d fc               mov   %edi,-0x4(%rbp)
40110d:    89 75 f8               mov   %esi,-0x8(%rbp)
401110:    89 55 f4               mov   %edx,-0xc(%rbp)
401113:    89 4d f0               mov   %ecx,-0x10(%rbp)
401116:    8b 55 fc               mov   -0x4(%rbp),%edx
401119:    8b 45 f8               mov   -0x8(%rbp),%eax
40111c:    01 c2                  add   %eax,%edx
40111e:    8b 45 f4               mov   -0xc(%rbp),%eax
401121:    01 c2                  add   %eax,%edx
401123:    8b 45 f0               mov   -0x10(%rbp),%eax
401126:    01 d0                  add   %edx,%eax
401128:    5d                     pop   %rbp
401129:    c3                     retq
```

1. **How and when the actual parameters are passed to the sum function :**

The answer of this question can be found from the below assembly code snippet taken from the above given assembly code of the main function.

```
401132:    c7 45 fc 02 00 00 00   movl  $0x2,-0x4(%rbp)     move 2 into stack at 0x4
401139:    c7 45 f8 03 00 00 00   movl  $0x3,-0x8(%rbp)     move 3 into stack at 0x8
401140:    c7 45 f4 05 00 00 00   movl  $0x5,-0xc(%rbp)     move 5 into stack at 0xc
401147:    c7 45 f0 07 00 00 00   movl  $0x7,-0x10(%rbp)    move 7 into stack at 0x10
```

40114e:	8b 4d f0	mov	-0x10(%rbp),%ecx	**move value from 0x10 to ecx**
401151:	8b 55 f4	mov	-0xc(%rbp),%edx	**move value from 0xc to edx**
401154:	8b 75 f8	mov	-0x8(%rbp),%esi	**move value from 0x8 to esi**
401157:	8b 45 fc	mov	-0x4(%rbp),%eax	
40115a:	89 c7	mov	%eax,%edi	**move value from 0x04 to edi**

We can see that the 4 integer variables namely x, y, z, w are first stored into the stack consecutively and then their values are loaded into 4 registers which are ecx, edx, esi, edi.

By examining the below assembly code snippet of the sum function, we can confirm that the parameters are passed to the sum function using registers and the values of all parameters are loaded into the registers before calling the sum function.

40110a:	89 7d fc	mov	%edi,-0x4(%rbp)	**move value from edi to 0x4**
40110d:	89 75 f8	mov	%esi,-0x8(%rbp)	**move value from esi to 0x8**
401110:	89 55 f4	mov	%edx,-0xc(%rbp)	**move value from edx to 0xc**
401113:	89 4d f0	mov	%ecx,-0x10(%rbp)	**move value from ecx to 0x10**

So, parameter values from the registers are stored at the consecutive locations in the stack frame of the sum function and then the operations are performed using the values stored in these memory locations.

2. **When and where main function copies the return address :**

In the chapter 9 and 10 of the book “Concepts of programming languages 10th edition”, it is mentioned that when a function calls another function then the address of the next instruction which immediately follows the call instruction, is copied into the stack. And when the called function is done executing, it uses that address to jump to that next instruction of the function which has called this function.

We can confirm this by examining the few top elements of the stack when we are inside the called function. The presence of the address of the next instruction of the callee function is enough to prove this fact.

So, we will mark 2 breakpoints. One at the main function and other at the sum function. Then we will start the debugging session of the gdb and run the sum.c program. After that, the execution will stop as soon as the control reaches the main function. Then we will use “nexti” instruction to execute one assembly instruction at a time until the “callq”(the function calling instruction) instruction is reached.

Just before the “callq” instruction, we will print the top 10 elements of stack using “x/10x $rsp” command to see that there is no return address in the stack at this point of execution and after the “callq” instruction is executed, we will again use this command to print the top 10 elements of the stack to confirm that now some return address is added into the stack.

Start the debugging session :

gdb sum

Add breakpoints at the main and sum function :

```
(gdb) b main
Breakpoint 1 at 0x401132: file sum.c, line 6.
(gdb) b sum
Breakpoint 2 at 0x401116: file sum.c, line 2.
```

Run the program :

```
(gdb) r
Starting program:
/home/chandrakishorsingh/Documents/iiit-allahabad/semester-1/programming-practices/practice
-sessions/1/sum

Breakpoint 1, main () at sum.c:6
6		int x = 2;
```

Run all the assembly instructions upto “callq” instruction(located at 40115c address) :

```
(gdb) nexti
7		int y = 3;
(gdb) nexti
8		int z = 5;
(gdb) nexti
9		int w = 7;
(gdb) nexti
10		int r = sum(x, y, z, w);
(gdb) nexti
0x0000000000401151	10		int r = sum(x, y, z, w);
(gdb) nexti
0x0000000000401154	10		int r = sum(x, y, z, w);
(gdb) nexti
0x0000000000401157	10		int r = sum(x, y, z, w);
(gdb) nexti
0x000000000040115a	10		int r = sum(x, y, z, w);
(gdb) nexti
0x000000000040115c10		int r = sum(x, y, z, w);
```

Examine the top 10 elements of the stack before jumping to the sum function :

```
(gdb) x/10x $rsp
0x7fffffffdcc0:  0x00000000   0x00000000   0x00401020   0x00000000
0x7fffffffdcd0:  0x00000007   0x00000005   0x00000003   0x00000002
0x7fffffffdce0:  0x00000000   0x00000000
```

The next instruction after the “callq” is “mov   %eax,-0x14(%rbp)” which is located at address **401161** but this address is not present in the top 10 elements of stack as seen from the above output.

So, we will continue executing the next instruction which is “callq” which will jump to the sum function.

Examine the top 10 elements of the stack after jumping to sum function :

```
(gdb) nexti

Breakpoint 2, sum (a=2, b=3, c=5, d=7) at sum.c:2
2                return a + b + c + d;
(gdb) x/10x $rsp
0x7fffffffdcb0:  0xffffdce0   0x00007fff   0x00401161   0x00000000
0x7fffffffdcc0:  0x00000000   0x00000000   0x00401020   0x00000000
0x7fffffffdcd0:  0x00000007   0x00000005
```

We can see that the address of the next instruction of the assembly version of main is now present in the stack.

From this, we can confirm that the return address of the callee function is stored in the stack just before the body of the called function starts executing.

2. Repeat question 1 first in C (using pointers) and later in C++ (by using reference variable) by making one of the parameters as pass by reference. Observe the change in the assembly version.

We will modify the sum.c program that we've used in the previous question to now pass one of the parameters as an integer pointer. Let's name this file as sum2.c

sum2.c file :

```
int sum(int a, int b, int c, int* d) {
	return a + b + c + *d;
}

int main() {
	int x = 2;
	int y = 3;
	int z = 5;
	int w = 7;
	int* p = &w;
	int r = sum(x, y, z, p);
	return 0;
}
```

Also, we will create an equivalent C++ program which will use reference instead of pointer. Let's name this file as sum2.cpp

sum2.cpp file :

```
int sum(int a, int b, int c, int& d) {
	return a + b + c + d;
}

int main() {
	int x = 2;
	int y = 3;
	int z = 5;
	int w = 7;
	int& p = w;
	int r = sum(x, y, z, p);
	return 0;
}
```

We will compile both of the files with debugging information using the following commands.

To compile sum2.c program :

gcc -g sum.c -O0 -o sum2c

To compile sum2.cpp program :

g++ -g sum.cpp -O0 -o sum2cpp

Let's examine the assembly code for the main and sum function of the sum.c file. We'll use "objdump -d sum2c" to get the assembly code for both of the functions(main and sum).

Assembly code for the main function is as follows.

```
000000000040112e <main>:
  40112e:	55                   	push   %rbp
  40112f:	48 89 e5             	mov    %rsp,%rbp
  401132:	48 83 ec 20          	sub    $0x20,%rsp
  401136:	c7 45 fc 02 00 00 00 	movl   $0x2,-0x4(%rbp)
  40113d:	c7 45 f8 03 00 00 00 	movl   $0x3,-0x8(%rbp)
  401144:	c7 45 f4 05 00 00 00 	movl   $0x5,-0xc(%rbp)
  40114b:	c7 45 e0 07 00 00 00 	movl   $0x7,-0x20(%rbp)
  401152:	48 8d 45 e0          	lea    -0x20(%rbp),%rax
  401156:	48 89 45 e8          	mov    %rax,-0x18(%rbp)
  40115a:	48 8b 4d e8          	mov    -0x18(%rbp),%rcx
  40115e:	8b 55 f4             	mov    -0xc(%rbp),%edx
  401161:	8b 75 f8             	mov    -0x8(%rbp),%esi
  401164:	8b 45 fc             	mov    -0x4(%rbp),%eax
  401167:	89 c7                	mov    %eax,%edi
  401169:	e8 98 ff ff ff       	callq  401106 <sum>
  40116e:	89 45 e4             	mov    %eax,-0x1c(%rbp)
  401171:	b8 00 00 00 00       	mov    $0x0,%eax
  401176:	c9                   	leaveq
  401177:	c3                   	retq
  401178:	0f 1f 84 00 00 00 00 	nopl   0x0(%rax,%rax,1)
  40117f:	00
```

Assembly code for the sum function is as follows.

```
0000000000401106 <sum>:
  401106:	55                   	push   %rbp
  401107:	48 89 e5             	mov    %rsp,%rbp
  40110a:	89 7d fc             	mov    %edi,-0x4(%rbp)
  40110d:	89 75 f8             	mov    %esi,-0x8(%rbp)
```

```
401110:   89 55 f4               mov    %edx,-0xc(%rbp)
401113:   48 89 4d e8            mov    %rcx,-0x18(%rbp)
401117:   8b 55 fc               mov    -0x4(%rbp),%edx
40111a:   8b 45 f8               mov    -0x8(%rbp),%eax
40111d:   01 c2                  add    %eax,%edx
40111f:   8b 45 f4               mov    -0xc(%rbp),%eax
401122:   01 c2                  add    %eax,%edx
401124:   48 8b 45 e8            mov    -0x18(%rbp),%rax
401128:   8b 00                  mov    (%rax),%eax
40112a:   01 d0                  add    %edx,%eax
40112c:   5d                pop    %rbp
40112d:   c3                retq
```

Examining how parameters are passed to sum function :

We will examine the below assembly code snippet from the assembly version of the main function.

```
401136:   c7 45 fc 02 00 00 00   movl   $0x2,-0x4(%rbp)       move 2 to add. 0x4
40113d:   c7 45 f8 03 00 00 00   movl   $0x3,-0x8(%rbp)       move 3 to add. 0x8
401144:   c7 45 f4 05 00 00 00   movl   $0x5,-0xc(%rbp)       move 5 to add. 0xc
40114b:   c7 45 e0 07 00 00 00   movl   $0x7,-0x20(%rbp)  move 7 to add. 0x20
401152:   48 8d 45 e0            lea   -0x20(%rbp),%rax     load the add. of  0x20 into rax
401156:   48 89 45 e8            mov    %rax,-0x18(%rbp)  move value of rax to add. 0x18
40115a:   48 8b 4d e8            mov    -0x18(%rbp),%rcx  move value at 0x18 to rcx
40115e:   8b 55 f4               mov    -0xc(%rbp),%edx   move value at 0xc to edx
401161:   8b 75 f8               mov    -0x8(%rbp),%esi   move value at 0x8 to esi
401164:   8b 45 fc               mov    -0x4(%rbp),%eax   move value at 0x4 to eax
401167:   89 c7                  mov    %eax,%edi         move value at eax to edi
```

From the above code, we can confirm that those parameters that are passed as value are copied into the registers(namely edi, esi, edx) and for the last parameter which was passed as the pointer, the address of the location where it is stored is loaded into register rcx and not the actual value(which is 7).

Examining the assembly version of C++ program using reference :

Now we'll examine the assembly version of sum2.cpp to find if there is any difference between its assembly code and the assembly code of sum2.c

We will use "objdump -d sum2cpp" to get the assembly code for the main and sum function.

Assembly code for the main function is as follows.

```
000000000040112e <main>:
  40112e:    55                      push   %rbp
  40112f:    48 89 e5                mov    %rsp,%rbp
  401132:    48 83 ec 20             sub    $0x20,%rsp
  401136:    c7 45 fc 02 00 00 00    movl   $0x2,-0x4(%rbp)
  40113d:    c7 45 f8 03 00 00 00    movl   $0x3,-0x8(%rbp)
  401144:    c7 45 f4 05 00 00 00    movl   $0x5,-0xc(%rbp)
  40114b:    c7 45 e0 07 00 00 00    movl   $0x7,-0x20(%rbp)
  401152:    48 8d 45 e0             lea    -0x20(%rbp),%rax
  401156:    48 89 45 e8             mov    %rax,-0x18(%rbp)
  40115a:    48 8b 4d e8             mov    -0x18(%rbp),%rcx
  40115e:    8b 55 f4                mov    -0xc(%rbp),%edx
  401161:    8b 75 f8                mov    -0x8(%rbp),%esi
  401164:    8b 45 fc                mov    -0x4(%rbp),%eax
  401167:    89 c7                   mov    %eax,%edi
  401169:    e8 98 ff ff ff          callq  401106 <_Z3sumiiiRi>
  40116e:    89 45 e4                mov    %eax,-0x1c(%rbp)
  401171:    b8 00 00 00 00          mov    $0x0,%eax
  401176:    c9                      leaveq
  401177:    c3                      retq
  401178:    0f 1f 84 00 00 00 00    nopl   0x0(%rax,%rax,1)
  40117f:    00
```

Assembly code for the sum function is as follows.

```
0000000000401106 <_Z3sumiiiRi>:
  401106:    55                      push   %rbp
  401107:    48 89 e5                mov    %rsp,%rbp
  40110a:    89 7d fc                mov    %edi,-0x4(%rbp)
  40110d:    89 75 f8                mov    %esi,-0x8(%rbp)
  401110:    89 55 f4                mov    %edx,-0xc(%rbp)
  401113:    48 89 4d e8             mov    %rcx,-0x18(%rbp)
  401117:    8b 55 fc                mov    -0x4(%rbp),%edx
  40111a:    8b 45 f8                mov    -0x8(%rbp),%eax
  40111d:    01 c2                   add    %eax,%edx
  40111f:    8b 45 f4                mov    -0xc(%rbp),%eax
  401122:    01 c2                   add    %eax,%edx
  401124:    48 8b 45 e8             mov    -0x18(%rbp),%rax
  401128:    8b 00                   mov    (%rax),%eax
  40112a:    01 d0                   add    %edx,%eax
  40112c:    5d                      pop    %rbp
  40112d:    c3                      retq
```

We can see that the assembly version of main and sum function for sum2.cpp is virtually the same as the assembly version of the main and sum function of sum2.c.

Hence, for this program, both the C and C++ version produce the same assembly code and there is virtually no difference between them.

3. How C/C++ compilers handle fixed stack dynamic and stack dynamic arrays?

How C/C++ compilers handle fixed stack dynamic arrays :

Fixed stack dynamic arrays are those whose size or number of elements of array, is already known at compile time and their memory allocation is done at run time.

We create a file namely fixed-stack-arr.c. It has the following code.

```
void func() {
	int arr[3] = {1, 2, 3};
}

int main() {
	func();
	return 0;
}
```

Then we compile the fixed-stack-arr.c with debugging information by the following command.

```
gcc fixed-stack-arr.c  -g -O0 -o fixed-stack-arr
```

And get the assembly code for the main and func function with the following command.

```
objdump -d fixed-stack-arr
```

The assembly code for the main function is as follows.

```
0000000000401122 <main>:
  401122:	55                   	push   %rbp
  401123:	48 89 e5             	mov    %rsp,%rbp
  401126:	b8 00 00 00 00       	mov    $0x0,%eax
  40112b:	e8 d6 ff ff ff       	callq  401106 <func>
  401130:	b8 00 00 00 00       	mov    $0x0,%eax
  401135:	5d                   	pop    %rbp
  401136:	c3                   	retq
  401137:	66 0f 1f 84 00 00 00 	nopw   0x0(%rax,%rax,1)
  40113e:	00 00
```

The assembly code for the func function is as follows.

```
0000000000401106 <func>:
  401106:	55                   	push   %rbp
  401107:	48 89 e5             	mov    %rsp,%rbp
```

```
40110a:    c7 45 f4 01 00 00 00   movl   $0x1,-0xc(%rbp)
401111:    c7 45 f8 02 00 00 00   movl   $0x2,-0x8(%rbp)
401118:    c7 45 fc 03 00 00 00   movl   $0x3,-0x4(%rbp)
40111f:    90                     nop
401120:    5d                     pop    %rbp
401121:    c3                     retq
```

We can see that the allocation for the array is done in the same manner as it would have done if we have declared three variables like int a = 1, b = 2, c = 3. Value of all of the 3 variables are stored in the stack frame.

Also the assembly code for the C++ version is as same as the assembly version of this C code.

How C/C++ compilers handle stack dynamic arrays :

Stack dynamic arrays are those whose size or number of elements are not known at compile time and their storage is also done at run allocation.

Note that, the standard C language does not support stack dynamic array but its support is added by many compilers including gcc.

We create a file namely stack-dynamic-arr.c. It has the following code.

```
void func(int n) {
	int arr[n];
	arr[0] = 1;
}

int main() {
	func(5);
	return 0;
}
```

Then we compile the stack-dynamic-arr.c with debugging information by the following command.

```
gcc -g stack-dynamic-arr.c -O0 -o stack-dynamic-arr-c
```

And get the assembly code for the main and func function with the following command.

```
objdump -d stack-dynamic-arr-c
```

The assembly code for the main function is as follows.

```
000000000040118a <main>:
```

```
 40118a:       55                   push   %rbp
 40118b:       48 89 e5             mov    %rsp,%rbp
 40118e:       bf 05 00 00 00       mov    $0x5,%edi
 401193:       e8 6e ff ff ff       callq  401106 <func>
 401198:       b8 00 00 00 00       mov    $0x0,%eax
 40119d:       5d                   pop    %rbp
 40119e:       c3                   retq
 40119f:       90                   nop
```

The assembly code for the func function is as follows.

```
0000000000401106 <func>:
 401106:       55                   push   %rbp
 401107:       48 89 e5             mov    %rsp,%rbp
 40110a:       48 83 ec 20          sub    $0x20,%rsp
 40110e:       89 7d ec             mov    %edi,-0x14(%rbp)
 401111:       48 89 e0             mov    %rsp,%rax
 401114:       48 89 c1             mov    %rax,%rcx
 401117:       8b 45 ec             mov    -0x14(%rbp),%eax
 40111a:       48 63 d0             movslq %eax,%rdx
 40111d:       48 83 ea 01          sub    $0x1,%rdx
 401121:       48 89 55 f8          mov    %rdx,-0x8(%rbp)
 401125:       48 63 d0             movslq %eax,%rdx
 401128:       49 89 d2             mov    %rdx,%r10
 40112b:       41 bb 00 00 00 00    mov    $0x0,%r11d
 401131:       48 63 d0             movslq %eax,%rdx
 401134:       49 89 d0             mov    %rdx,%r8
 401137:       41 b9 00 00 00 00    mov    $0x0,%r9d
 40113d:       48 98                cltq
 40113f:       48 8d 14 85 00 00 00 lea    0x0(,%rax,4),%rdx
 401146:       00
 401147:       b8 10 00 00 00       mov    $0x10,%eax
 40114c:       48 83 e8 01          sub    $0x1,%rax
 401150:       48 01 d0             add    %rdx,%rax
 401153:       be 10 00 00 00       mov    $0x10,%esi
 401158:       ba 00 00 00 00       mov    $0x0,%edx
 40115d:       48 f7 f6             div    %rsi
 401160:       48 6b c0 10          imul   $0x10,%rax,%rax
 401164:       48 29 c4             sub    %rax,%rsp
 401167:       48 89 e0             mov    %rsp,%rax
 40116a:       48 83 c0 03          add    $0x3,%rax
 40116e:       48 c1 e8 02          shr    $0x2,%rax
 401172:       48 c1 e0 02          shl    $0x2,%rax
 401176:       48 89 45 f0          mov    %rax,-0x10(%rbp)
```

```
40117a:    48 8b 45 f0             mov    -0x10(%rbp),%rax
40117e:    c7 00 01 00 00 00       movl   $0x1,(%rax)
401184:    48 89 cc                mov    %rcx,%rsp
401187:    90                      nop
401188:    c9                      leaveq
401189:    c3                      retq
```

The assembly code for the func function is quite long. After searching for some time, I came to know that most of the instructions are present to check for overflow of the stack.

4. Create more than one heap dynamic variables in C/C++ and observe the difference in addresses of different heap dynamic variables and also compare them with static and stack dynamic variables.

Heap dynamic variables are those for which memory is allocated at run time using some special memory allocation construct of the programming language. For ex. C uses functions like malloc(), calloc() etc. to create heap dynamic variables while C++ also has the “new” keyword to do this.

All heap dynamic variables are stored in the heap area of the memory. Unlike stack memory, storage of heap variables may or may not be in contiguous locations of heap.

To confirm this we will create a program in C++ which will have two heap dynamic variables and then we will find their locations to see that they are not stored consecutively.

We create a C++ source file called heap-dynamic.cpp. It has the following code.

```
int main() {
	int* a = new int[3];
	int* b = new int[5];
	return 0;
}
```

Then we compile the heap-dynamic.c with debugging information by the following command.

```
g++ -g heap-dynamic.cpp -O0 -o heap-dynamic
```

And get the assembly code for the main function with the following command.

```
objdump -d heap-dynamic
```

The assembly code for the main function is as follows.

```
0000000000401126 <main>:
  401126:	55                	push   %rbp
  401127:	48 89 e5          	mov    %rsp,%rbp
  40112a:	48 83 ec 10       	sub    $0x10,%rsp
  40112e:	bf 0c 00 00 00    	mov    $0xc,%edi
  401133:	e8 f8 fe ff ff    	callq  401030 <_Znam@plt>
  401138:	48 89 45 f8       	mov    %rax,-0x8(%rbp)
  40113c:	bf 14 00 00 00    	mov    $0x14,%edi
  401141:	e8 ea fe ff ff    	callq  401030 <_Znam@plt>
  401146:	48 89 45 f0       	mov    %rax,-0x10(%rbp)
```

```
40114a:    b8 00 00 00 00        mov    $0x0,%eax
40114f:    c9                    leaveq
401150:    c3                    retq
401151:    66 2e 0f 1f 84 00 00  nopw   %cs:0x0(%rax,%rax,1)
401158:    00 00 00
40115b:    0f 1f 44 00 00        nopl   0x0(%rax,%rax,1)
```

The instruction “callq 401030 <_Znam@plt>” is responsible for allocating the space for the variables a and b. The address of the newly created heap dynamic variable is loaded into the rax register by “callq 401030 <_Znam@plt>” instruction. This address is then loaded into the position $rpb - 8(for variable a) and and $rpb - 10(for variable b) positions.

We can check their address using the command “x &a” and “x &b”. To do that, first we will create a breakpoint at main function and then execute the next 2 lines. The output of these steps is as follows.

```
gdb heap-dynamic

(gdb) b main
Breakpoint 1 at 0x40112e: file heap-dynamic.cpp, line 2.

(gdb) r
Starting program:
/home/chandrakishorsingh/Documents/iiit-allahabad/semester-1/programming-practices/practice
-sessions/4/heap-dynamic

Breakpoint 1, main () at heap-dynamic.cpp:2
2           int* a = new int[3];
(gdb) n
3           int* b = new int[5];
(gdb) n
4           return 0;
(gdb) x &a
0x7fffffffdcc8:  0x00416eb0
(gdb) x &b
0x7fffffffdcc0:  0x00416ed0
(gdb)
```

From the address of variables a and b, it is clear that in heap, variables may or may not be stored in the consecutive memory locations. This is the main difference between heap dynamic and stack dynamic variables.

The main aspect in which heap dynamic variables differ from static variables is that the address of static variable is binded to memory location at the loading phase of program and they can't be deleted unlike heap dynamic variables which can be deleted in C++ by "delete" keyword.

5. On page number 359 of book Concepts of programming languages 10th edition, Prof. Sebesta has discussed three different strategies to implement a switch statement depending on the range of case constants. Find out which method is used by C/C++ in implementing switch statement. Try using different range of case constants to observe if the compiler selects different strategy depending on range of case constants.

The three strategies that were discussed are as follows.

1. Linear Search :

When the number of cases is less than 10 then the compiler uses linear search.

2. Hash Table :

When the number of cases is more than 10, then a hash table can be used. But this won't be a good approach if the language allows range of values for the case expression.

3. Using array :

When the range of case values are small and more than half of the values lies inside the range, then using an array whose index represents the values and elements contain the address of branch is used.

We will use the below C++ program called age-group.cpp to demonstrate the strategy used by g++ compiler.

age-group.cpp file :

```
int main() {
	int age = 10;
	int result = -1;
	switch (age) {
		case 10:
			result = 1;
			break;
		case 20:
			result = 2;
			break;

		case 30:
			result = 3;
			break;
	}
}
```

Then we compile the age-group.cpp with debugging information by the following command.

g++ -g age-group.cpp -O0 -o age-group

And get the assembly code for the main function with the following command.

objdump -d age-group

The assembly code for the main function is as follows.

```
0000000000401106 <main>:
  401106:       55                      push   %rbp
  401107:       48 89 e5                mov    %rsp,%rbp
  40110a:       c7 45 fc 0a 00 00 00    movl   $0xa,-0x4(%rbp)
  401111:       c7 45 f8 ff ff ff ff    movl   $0xffffffff,-0x8(%rbp)
  401118:       83 7d fc 1e             cmpl   $0x1e,-0x4(%rbp)
  40111c:       74 26                   je     401144 <main+0x3e>
  40111e:       83 7d fc 1e             cmpl   $0x1e,-0x4(%rbp)
  401122:       7f 28                   jg     40114c <main+0x46>
  401124:       83 7d fc 0a             cmpl   $0xa,-0x4(%rbp)
  401128:       74 08                   je     401132 <main+0x2c>
  40112a:       83 7d fc 14             cmpl   $0x14,-0x4(%rbp)
  40112e:       74 0b                   je     40113b <main+0x35>
  401130:       eb 1a                   jmp    40114c <main+0x46>
  401132:       c7 45 f8 01 00 00 00    movl   $0x1,-0x8(%rbp)
  401139:       eb 11                   jmp    40114c <main+0x46>
  40113b:       c7 45 f8 02 00 00 00    movl   $0x2,-0x8(%rbp)
  401142:       eb 08                   jmp    40114c <main+0x46>
  401144:       c7 45 f8 03 00 00 00    movl   $0x3,-0x8(%rbp)
  40114b:       90                      nop
  40114c:       b8 00 00 00 00          mov    $0x0,%eax
  401151:       5d                      pop    %rbp
  401152:       c3                      retq
  401153:       66 2e 0f 1f 84 00 00    nopw   %cs:0x0(%rax,%rax,1)
  40115a:       00 00 00
  40115d:       0f 1f 00                nopl   (%rax)
```

We can see that the values 10, 20, 30 are actually hardcoded into the instructions itself. And hence none of the above mentioned strategies is used. This is because every compiler does not follows all the strategies mentioned in the book.

6. Comment on how C/C++ compiler uses stack to implement a recursive program. Whether it uses the method described in the text book on page numbers 351 − 353?

We will use below recursive C++ program to find out how the recursive program is implemented.

fact.cpp file :

```
int fibo(int n) {
	if (n == 1 || n == 0)
		return 1;
	return fibo(n - 1) + fibo(n - 2);
}

int main() {
	int r = fibo(5);
	return 0;
}
```

We compile the fact.cpp with debugging information by the following command.

```
g++ -g fibo.cpp -O0 -o fibo
```

And get the assembly code for the main and fact function with the following command.

```
objdump -d fibo
```

The assembly code for the main function is as follows.

```
0000000000401149 <main>:
  401149:	55                   	push   %rbp
  40114a:	48 89 e5             	mov    %rsp,%rbp
  40114d:	b8 00 00 00 00       	mov    $0x0,%eax
  401152:	5d                   	pop    %rbp
  401153:	c3                   	retq
  401154:	66 2e 0f 1f 84 00 00 	nopw   %cs:0x0(%rax,%rax,1)
  40115b:	00 00 00
  40115e:	66 90                	xchg   %ax,%ax
```

The assembly code for the fibo function is as follows.

```
0000000000401106 <_Z4fiboi>:
  401106:	55                   	push   %rbp
  401107:	48 89 e5             	mov    %rsp,%rbp
  40110a:	53                   	push   %rbx
```

```
40110b:    48 83 ec 18       sub    $0x18,%rsp
40110f:    89 7d ec          mov    %edi,-0x14(%rbp)
401112:    83 7d ec 01       cmpl   $0x1,-0x14(%rbp)
401116:    74 06             je     40111e <_Z4fiboi+0x18>
401118:    83 7d ec 00       cmpl   $0x0,-0x14(%rbp)
40111c:    75 07             jne    401125 <_Z4fiboi+0x1f>
40111e:    b8 01 00 00 00    mov    $0x1,%eax
401123:    eb 1e             jmp    401143 <_Z4fiboi+0x3d>
401125:    8b 45 ec          mov    -0x14(%rbp),%eax
401128:    83 e8 01          sub    $0x1,%eax
40112b:    89 c7             mov    %eax,%edi
40112d:    e8 d4 ff ff ff    callq  401106 <_Z4fiboi>
401132:    89 c3             mov    %eax,%ebx
401134:    8b 45 ec          mov    -0x14(%rbp),%eax
401137:    83 e8 02          sub    $0x2,%eax
40113a:    89 c7             mov    %eax,%edi
40113c:    e8 c5 ff ff ff    callq  401106 <_Z4fiboi>
401141:    01 d8             add    %ebx,%eax
401143:    48 8b 5d f8       mov    -0x8(%rbp),%rbx
401147:    c9          leaveq
401148:    c3          retq
```

We can see that the parameter 5 is passed to the register edi. Inside the fibo function, the value of parameter is stored at the location $rpb - 0x14. After that, the value of parameter is compared to 1 and 0 in the instruction stored at 401112 and 401118 respectively. The function fibo is again called until the parameter value becomes 1 or 0.