PPL ASSIGNMENT
SUBMITTED BY :
Jyotika Bhatti
IIT2019036

Q1. 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.

Sol.
The given code is of C program with 3 formal parameters a, b and c passed from main function and the "function" is called .

By default, in c language the functions are called by value .
For the below program, we have values that are passed by (formal parameters), Which means that the variables a,b, c, in the method "function" are the actual parameters and when they are called from the main function, they are the formal parameters.

```
#include<stdio.h>

int function(int a , int b, int c)
{
        int product;
        product = a*b*c;

        return product;

}
int main()
{
        int a=2;
        int b=3;
        int c=4;

        function(a, b, c);
        return 0;

}
```

The corresponding assembly code of the “function” is given by :

```
	.file	"ques1.c"
	.text
	.globl	function
	.type	function, @function
function:
.LFB0:
	.cfi_startproc
	endbr64
	pushq	%rbp
	.cfi_def_cfa_offset 16
	.cfi_offset 6, -16
	movq	%rsp, %rbp
	.cfi_def_cfa_register 6
	movl	%edi, -20(%rbp)
	movl	%esi, -24(%rbp)
	movl	%edx, -28(%rbp)
	movl	-20(%rbp), %eax
	imull	-24(%rbp), %eax
	movl	-28(%rbp), %edx
	imull	%edx, %eax
	movl	%eax, -4(%rbp)
	movl	-4(%rbp), %eax
	popq	%rbp
	.cfi_def_cfa 7, 8
	ret
	.cfi_endproc
.LFE0:
	.size	function, .-function
	.globl	main
	.type	main, @function
main:
```

The method “function” have the values of actual parameters from the formal parameters from the main function.Then the values of actual parameters getting passed into the formal parameters. Then the main function copies the addresses of the variables and passes to the calle function,

```
.LFE0:
        .size   function, .-function
        .globl  main
        .type   main, @function
main:
.LFB1:
        .cfi_startproc
        endbr64
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        subq    $16, %rsp
        movl    $2, -12(%rbp)
        movl    $3, -8(%rbp)
        movl    $4, -4(%rbp)
        movl    -4(%rbp), %edx
        movl    -8(%rbp), %ecx
        movl    -12(%rbp), %eax
        movl    %ecx, %esi
        movl    %eax, %edi
        call    function
        movl    $0, %eax
        leave
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE1:
        .size   main, .-main
        .ident  "GCC: (Ubuntu 10.2.0-13ubuntu1) 10.2.0"
```

Now, we have to find out how and when values of actual parameters are passed to the formal parameters in the called "function".

Now, from the assembly code of the above program,

```
main:
.LFB1:
	.cfi_startproc
	endbr64
	pushq	%rbp
	.cfi_def_cfa_offset 16
	.cfi_offset 6, -16
	movq	%rsp, %rbp
	.cfi_def_cfa_register 6
	subq	$16, %rsp
	movl	$2, -12(%rbp)
	movl	$3, -8(%rbp)
	movl	$4, -4(%rbp)
	movl	-4(%rbp), %edx
	movl	-8(%rbp), %ecx
	movl	-12(%rbp), %eax
	movl	%ecx, %esi
	movl	%eax, %edi
	call	function
	movl	$0, %eax
	leave
	.cfi_def_cfa 7, 8
	ret
	.cfi_endproc
```

The highlighted part of the code stores the value of RBP in the stack , which saves the value of RSP into RBP . It allocates about 16 bytes to the stack since it will be a space for the storage of local variables and temporaries .

```
function:
.LFB0:
        .cfi_startproc
        endbr64
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movl    %edi, -20(%rbp)
        movl    %esi, -24(%rbp)
        movl    %edx, -28(%rbp)
        movl    -20(%rbp), %eax
        imull   -24(%rbp), %eax
        movl    -28(%rbp), %edx
        imull   %edx, %eax
        movl    %eax, -4(%rbp)
        movl    -4(%rbp), %eax
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   function, .-function
        .globl  main
        .type   main, @function
```

Similarly , the assembly code of the method "function" depicts storing the value of RBP in the stack , hence saving the value of RSP into RBP .

```
#include<stdio.h>

int function(int a , int b, int c)
{
        int product;
        product = a*b*c;

        return product;

}
int main()
{
        int a=2;
        int b=3;
        int c=4;

        function(a, b, c);
        return 0;

}
```

For the actual parameters that are passed, to call for method "function" , The values of these parameters are stored into the save registers which are depicted in their respective callers.

```
function:
.LFB0:
        .cfi_startproc
        endbr64
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movl    %edi, -20(%rbp)
        movl    %esi, -24(%rbp)
        movl    %edx, -28(%rbp)
        movl    -20(%rbp), %eax
        imull   -24(%rbp), %eax
        movl    -28(%rbp), %edx
        imull   %edx, %eax
        movl    %eax, -4(%rbp)
        movl    -4(%rbp), %eax
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
```

```
main:
.LFB1:
        .cfi_startproc
        endbr64
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        subq    $16, %rsp
        movl    $2, -12(%rbp)
        movl    $3, -8(%rbp)
        movl    $4, -4(%rbp)
        movl    -4(%rbp), %edx
        movl    -8(%rbp), %ecx
        movl    -12(%rbp), %eax
        movl    %ecx, %esi
        movl    %eax, %edi
        call    function
        movl    $0, %eax
        leave
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
```

Now, if we set up a breakpoint on the main function ,
The reference address of the variable will be the same as passed into the actual parameter. Which could be easily verified by using gdb debugger , by using the following steps.

```
adminhp@adminhp-G3-3579:~/Desktop/SEM4/PPL/ASSIGNMENT/Q1$ CFLAGS="-g -O0" make prog
cc -g -O0    prog.c   -o prog
adminhp@adminhp-G3-3579:~/Desktop/SEM4/PPL/ASSIGNMENT/Q1$ gdb prog
GNU gdb (Ubuntu 9.2-0ubuntu2) 9.2
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from prog...
(gdb) break main
Breakpoint 1 at 0x114f: file prog.c, line 12.
(gdb) run
Starting program: /home/adminhp/Desktop/SEM4/PPL/ASSIGNMENT/Q1/prog

Breakpoint 1, main () at prog.c:12
12      {
(gdb) next 4
17              function(a, b, c);
(gdb) disassemble
```

```
(gdb) disassemble
Dump of assembler code for function main:
   0x000055555555514f <+0>:     endbr64
   0x0000555555555153 <+4>:     push   %rbp
   0x0000555555555154 <+5>:     mov    %rsp,%rbp
   0x0000555555555157 <+8>:     sub    $0x10,%rsp
   0x000055555555515b <+12>:    movl   $0x2,-0xc(%rbp)
   0x0000555555555162 <+19>:    movl   $0x3,-0x8(%rbp)
   0x0000555555555169 <+26>:    movl   $0x4,-0x4(%rbp)
=> 0x0000555555555170 <+33>:    mov    -0x4(%rbp),%edx
   0x0000555555555173 <+36>:    mov    -0x8(%rbp),%ecx
   0x0000555555555176 <+39>:    mov    -0xc(%rbp),%eax
   0x0000555555555179 <+42>:    mov    %ecx,%esi
   0x000055555555517b <+44>:    mov    %eax,%edi
   0x000055555555517d <+46>:    callq  0x555555555129 <function>
   0x0000555555555182 <+51>:    mov    $0x0,%eax
   0x0000555555555187 <+56>:    leaveq
   0x0000555555555188 <+57>:    retq
End of assembler dump.
```

```
(gdb) x &a
0x7fffffffdf64: 0x00000002
(gdb) x &b
0x7fffffffdf68: 0x00000003
(gdb) x &c
0x7fffffffdf6c: 0x00000004
```

Similarly , when we pass the addresses of a , b and c which are at -4(%RBP), -8(%RBP) AND -12(%RBP) will also give the same result.

```
(gdb) x $rbp - 8
0x7fffffffdf68: 0x00000003
(gdb) x $rbp - 4
0x7fffffffdf6c: 0x00000004
(gdb) x $rbp - 12
0x7fffffffdf64: 0x00000002
```

Hence, the reference address of the actual parameters stores their actual values at these parameters .

Now, discussing where the values of the actual parameters in caller function get stored in the local variables of the callee function by the callee save registers.

Q2.Repeat question 1 first in C (using pointers) and later in C++ (by using reference variable) by making one of the parameters pass by reference. Observe the change in the assembly version.

The program in ques 1 , is amended as ques2.c and ques2.cpp which passes one of the parameters by reference.

Now, when we call a function when its parameters are passed by reference, the operation that is basically performed upon formal parameters, affects the actual parameters.
Since the operations are performed upon the value that is stored in the address of the actual parameter

The C program is followed as :

```
#include<stdio.h>

int function(int a , int b, int *c)
{
        int product;
        product = a * b * *c;

        return product;

}
int main()
{
        int a=2;
        int b=3;
        int c=4;

        function(a, b, &c);

        return 0;

}
```

The corresponding Assembly code is given as :

```
        .file   "ques2.c"
        .text
        .globl  function
        .type   function, @function
function:
.LFB0:
        .cfi_startproc
        endbr64
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movl    %edi, -20(%rbp)
        movl    %esi, -24(%rbp)
        movq    %rdx, -32(%rbp)
        movl    -20(%rbp), %eax
        imull   -24(%rbp), %eax
        movl    %eax, %edx
        movq    -32(%rbp), %rax
        movl    (%rax), %eax
        imull   %edx, %eax
        movl    %eax, -4(%rbp)
        movl    -4(%rbp), %eax
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   function, .-function
        .globl  main
        .type   main, @function
main:
.LFB1:
        .cfi_startproc
        endbr64
        pushq   %rbp
        .cfi_def_cfa_offset 16
```

In the method “function” , where the values are passed by reference pointers , The memory is allocated and the canary gets created which pushes that into the stack and further erases it. This means that we store the program into its memory address , hence the canary value gets stored into the stack until it is going within the memory limit.

```
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   function, .-function
        .globl  main
        .type   main, @function
main:
.LFB1:
        .cfi_startproc
        endbr64
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        subq    $32, %rsp
        movq    %fs:40, %rax
        movq    %rax, -8(%rbp)
        xorl    %eax, %eax
        movl    $2, -16(%rbp)
        movl    $3, -12(%rbp)
        movl    $4, -20(%rbp)
        leaq    -20(%rbp), %rdx
        movl    -12(%rbp), %ecx
        movl    -16(%rbp), %eax
        movl    %ecx, %esi
        movl    %eax, %edi
        call    function
        movl    $0, %eax
        movq    -8(%rbp), %rsi
        subq    %fs:40, %rsi
        je      .L5
        call    __stack_chk_fail@PLT
.L5:
        leave
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE1:
        .size   main, .-main
        .ident  "GCC: (Ubuntu 10.2.0-13ubuntu1) 10.2.0"
```

In the main method,
The value of a , b and c are stored as 2, 3 , and 4 , then stored into the save registers at their corresponding addresses of 16(%RBP), -12(%RBP) and -20(%RBP).

The corresponding C++ program, where we pass a variable by reference is followed as :

```cpp
#include<iostream>
using namespace std;

int function(int a , int b, int& c)
{
        int product;
        product = a*b*c;

        return product;

}
int main()
{
        int a=2;
        int b=3;
        int c=4;

        function(a, b, c);
        return 0;

}
```

The assembly code of the given program in C++ is given by :

```
_ZStL19piecewise_construct:
        .zero   1
        .local  _ZStL8__ioinit
        .comm   _ZStL8__ioinit,1,1
        .text
        .globl  _Z8functioniiRi
        .type   _Z8functioniiRi, @function
_Z8functioniiRi:
.LFB1572:
        .cfi_startproc
        endbr64
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movl    %edi, -20(%rbp)
        movl    %esi, -24(%rbp)
        movq    %rdx, -32(%rbp)
        movl    -20(%rbp), %eax
        imull   -24(%rbp), %eax
        movl    %eax, %edx
        movq    -32(%rbp), %rax
        movl    (%rax), %eax
        imull   %edx, %eax
        movl    %eax, -4(%rbp)
        movl    -4(%rbp), %eax
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE1572:
        .size   _Z8functioniiRi, .-_Z8functioniiRi
        .globl  main
        .type   main, @function
main:
```

```
	ret
	.cfi_endproc
.LFE1572:
	.size	_Z8functioniiRi, .-_Z8functioniiRi
	.globl	main
	.type	main, @function
main:
.LFB1573:
	.cfi_startproc
	endbr64
	pushq	%rbp
	.cfi_def_cfa_offset 16
	.cfi_offset 6, -16
	movq	%rsp, %rbp
	.cfi_def_cfa_register 6
	subq	$32, %rsp
	movq	%fs:40, %rax
	movq	%rax, -8(%rbp)
	xorl	%eax, %eax
	movl	$2, -16(%rbp)
	movl	$3, -12(%rbp)
	movl	$4, -20(%rbp)
	leaq	-20(%rbp), %rdx
	movl	-12(%rbp), %ecx
	movl	-16(%rbp), %eax
	movl	%ecx, %esi
	movl	%eax, %edi
	call	_Z8functioniiRi
	movl	$0, %eax
	movq	-8(%rbp), %rsi
	subq	%fs:40, %rsi
	je	.L5
	call	__stack_chk_fail@PLT
.L5:
	leave
	.cfi_def_cfa 7, 8
	ret
```

The values a=2, b=3, c=4 are stored in stack at locations -16(%RBP), -12(%RBP) and -20(%RBP) , which are allocated as registers rdx, ecx, eax respectively.
Then the function call is made to the method “function” which stores the corresponding values of a , b , and c at save registers -20(%RBP), -24(%RBP) and -32(%RBP) after dereferencing the references variable then the final product is calculated and returned from there back to the main function .
This whole value gets stored at EAX and stored , then returned back .
Then hence the updated value of the product gets updated in the main function, therefore the result is computed

Q3. How C/C++ compilers handle fixed stack dynamic and stack dynamic arrays?

```c
#include<stdio.h>

void fixedstackdynamic()
{
        int a[10];   // allocated fixed stack dynamically
                    // this means that size of array is known at compile time
}

void stackdynamic(int n)
{
        int a[n];  // allocated stack dynamically
                   // the size of the array is unknown at runtime
                   // hence the memory will also be allocated  at runtime only
}

int main()
{

        fixedstackdynamic();
        stackdynamic(10);

        return 0;
}
```

The assembly code is given as :

```
        .file   "ques3.c"
        .text
        .globl  fixedstackdynamic
        .type   fixedstackdynamic, @function
fixedstackdynamic:
.LFB0:
        .cfi_startproc
        endbr64
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        subq    $48, %rsp
        movq    %fs:40, %rax
        movq    %rax, -8(%rbp)
        xorl    %eax, %eax
        nop
        movq    -8(%rbp), %rax
        subq    %fs:40, %rax
        je      .L2
        call    __stack_chk_fail@PLT
.L2:
        leave
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
```

```
stackdynamic:
.LFB1:
        .cfi_startproc
        endbr64
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        pushq   %rbx
        subq    $56, %rsp
        .cfi_offset 3, -24
        movl    %edi, -52(%rbp)
        movq    %fs:40, %rax
        movq    %rax, -24(%rbp)
        xorl    %eax, %eax
        movq    %rsp, %rax
        movq    %rax, %rsi
        movl    -52(%rbp), %eax
        movslq  %eax, %rdx
        subq    $1, %rdx
        movq    %rdx, -40(%rbp)
        movslq  %eax, %rdx
        movq    %rdx, %r8
        movl    $0, %r9d
        movslq  %eax, %rdx
        movq    %rdx, %rcx
        movl    $0, %ebx
        cltq
        leaq    0(,%rax,4), %rdx
        movl    $16, %eax
        subq    $1, %rax
        addq    %rdx, %rax
        movl    $16, %ebx
        movl    $0, %edx
        divq    %rbx
        imulq   $16, %rax, %rax
        movq    %rax, %rcx
        andq    $-4096, %rcx
        movq    %rsp, %rdx
        subq    %rcx, %rdx
```

We know that the function "fixedstackdynamic" , the array is allocated fixed stack dynamically which means that the size of the array gets allocated at compile time , but the memory is allocated to it at runtime.

Also, in the function "stackdynamic" , the array is allocated stack dynamically which means that the size of the array is unknown at compile time and this is known at runtime only and subsequently the memory will also be allocated at runtime.

In the function "stack dynamic" , the length of the array is present in EDI and gets stored in $RBP . further there are a few lines of code for the buffer stack overflow.

When the subprograms are executed, are also unbound from storage when the execution gets terminated.
In c and c++, the local variables are static dynamic unless the function is declared static, although in most of the contemporary languages, the local variables in the subprograms are by default stack dynamic.

Now, talking about the advantage that stack dynamic have is, that the local variables are stored in an active subprogram which could be shared with the local variables in the other inactive subprograms.
But stack dynamic, also has some disadvantages like the cost of time to allocate , initialise and deallocate is very expensive.

When all the local variables are stack dynamic, they cannot retain the values of data of local variables between these calls.

```
        movl    $16, %ebx
        movl    $0, %edx
        divq    %rbx
        imulq   $16, %rax, %rax
        movq    %rax, %rcx
        andq    $-4096, %rcx
        movq    %rsp, %rdx
        subq    %rcx, %rdx
.L4:
        cmpq    %rdx, %rsp
        je      .L5
        subq    $4096, %rsp
        orq     $0, 4088(%rsp)
        jmp     .L4
.L5:
        movq    %rax, %rdx
        andl    $4095, %edx
        subq    %rdx, %rsp
        movq    %rax, %rdx
        andl    $4095, %edx
        testq   %rdx, %rdx
        je      .L6
        andl    $4095, %eax
        subq    $8, %rax
        addq    %rsp, %rax
        orq     $0, (%rax)
.L6:
        movq    %rsp, %rax
        addq    $3, %rax
        shrq    $2, %rax
        salq    $2, %rax
        movq    %rax, -32(%rbp)
        movq    %rsi, %rsp
        nop
        movq    -24(%rbp), %rax
        subq    %fs:40, %rax
        je      .L7
        call    __stack_chk_fail@PLT
.L7:
        movq    -8(%rbp), %rbx
        leave
        .cfi_def_cfa 7, 8
```

```
main:
.LFB2:
        .cfi_startproc
        endbr64
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movl    $0, %eax
        call    fixedstackdynamic
        movl    $10, %edi
        call    stackdynamic
        movl    $0, %eax
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
```

In the above assembly code, nowhere could it be seen that the array is getting allocated some memory but not the allocation of array.
In the main function, the value that is passed to staticdynamic() funation is stored in EDI .

Q4.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.

```
#include<iostream>
using namespace std;
int main()
{
        int * f = new int;
        int * s = new int;

        *f = 15;
        *s = 16;

        cout<< *f + *s;

        delete f;
        delete s;

        return 0;
}
```

Here, I have declared two heap variables as pointers and the values 15 and 16 are assigned to them respectively.
The assembly code of the following program is given as :

```
main:
.LFB1572:
        .cfi_startproc
        endbr64
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        subq    $16, %rsp
        movl    $4, %edi
        call    _Znwm@PLT
        movq    %rax, -16(%rbp)
        movl    $4, %edi
        call    _Znwm@PLT
        movq    %rax, -8(%rbp)
        movq    -16(%rbp), %rax
        movl    $15, (%rax)
        movq    -8(%rbp), %rax
        movl    $16, (%rax)
        movq    -16(%rbp), %rax
        movl    (%rax), %edx
        movq    -8(%rbp), %rax
        movl    (%rax), %eax
        addl    %edx, %eax
        movl    %eax, %esi
        leaq    _ZSt4cout(%rip), %rdi
        call    _ZNSolsEi@PLT
        movq    -16(%rbp), %rax
        testq   %rax, %rax
        je      .L2
        movl    $4, %esi
        movq    %rax, %rdi
        call    _ZdlPvm@PLT
.L2:
        movq    -8(%rbp), %rax
        testq   %rax, %rax
        je      .L3
        movl    $4, %esi
        movq    %rax, %rdi
        call    _ZdlPvm@PLT
.L3:
        movl    $0, %eax
```

Also, the address of variables f and s , is seen that unlike the stack , the variables which are stored , are done on non-consecutive memory addresses as the heap will act as a pool of storage which allocates the memory very randomly.

```
Breakpoint 1, 0x00005555555551c9 in main ()
(gdb) disassemble
Dump of assembler code for function main:
=> 0x00005555555551c9 <+0>:     endbr64
   0x00005555555551cd <+4>:     push   %rbp
   0x00005555555551ce <+5>:     mov    %rsp,%rbp
   0x00005555555551d1 <+8>:     sub    $0x10,%rsp
   0x00005555555551d5 <+12>:    mov    $0x4,%edi
   0x00005555555551da <+17>:    callq  0x5555555550a0 <_Znwm@plt>
   0x00005555555551df <+22>:    mov    %rax,-0x10(%rbp)
   0x00005555555551e3 <+26>:    mov    $0x4,%edi
   0x00005555555551e8 <+31>:    callq  0x5555555550a0 <_Znwm@plt>
   0x00005555555551ed <+36>:    mov    %rax,-0x8(%rbp)
   0x00005555555551f1 <+40>:    mov    -0x10(%rbp),%rax
   0x00005555555551f5 <+44>:    movl   $0xf,(%rax)
   0x00005555555551fb <+50>:    mov    -0x8(%rbp),%rax
   0x00005555555551ff <+54>:    movl   $0x10,(%rax)
   0x0000555555555205 <+60>:    mov    -0x10(%rbp),%rax
   0x0000555555555209 <+64>:    mov    (%rax),%edx
   0x000055555555520b <+66>:    mov    -0x8(%rbp),%rax
   0x000055555555520f <+70>:    mov    (%rax),%eax
   0x0000555555555211 <+72>:    add    %edx,%eax
   0x0000555555555213 <+74>:    mov    %eax,%esi
   0x0000555555555215 <+76>:    lea    0x2e24(%rip),%rdi        # 0x555555558040 <_ZSt4cout@@GLIBCXX_3.4>
--Type <RET> for more, q to quit, c to continue without paging--c
   0x000055555555521c <+83>:    callq  0x5555555550d0 <_ZNSolsEi@plt>
   0x0000555555555221 <+88>:    mov    -0x10(%rbp),%rax
   0x0000555555555225 <+92>:    test   %rax,%rax
   0x0000555555555228 <+95>:    je     0x555555555237 <main+110>
   0x000055555555522a <+97>:    mov    $0x4,%esi
   0x000055555555522f <+102>:   mov    %rax,%rdi
   0x0000555555555232 <+105>:   callq  0x5555555550b0 <_ZdlPvm@plt>
   0x0000555555555237 <+110>:   mov    -0x8(%rbp),%rax
   0x000055555555523b <+114>:   test   %rax,%rax
   0x000055555555523e <+117>:   je     0x55555555524d <main+132>
   0x0000555555555240 <+119>:   mov    $0x4,%esi
   0x0000555555555245 <+124>:   mov    %rax,%rdi
   0x0000555555555248 <+127>:   callq  0x5555555550b0 <_ZdlPvm@plt>
   0x000055555555524d <+132>:   mov    $0x0,%eax
   0x0000555555555252 <+137>:   leaveq
   0x0000555555555253 <+138>:   retq
End of assembler dump.
```

When a dynamically allocated variable is deleted, the memory is "returned" to the heap and can then be reassigned as future allocation requests are received.

From the above code, we could observe how the value of each variable is compared and uses the further jump statements after comparison.

Here, .L2, .L3 and .L4 loads the addresses of the location of the corresponding .LC0 and .LC1 , now in case there is no value that is matched then

Deleting a pointer does not delete the variable, it just returns the memory at the associated address back to the operating system.

```
movl    $4, %edi
call    _Znwm@PLT
movq    %rax, -16(%rbp)
movl    $4, %edi
call    _Znwm@PLT
movq    %rax, -8(%rbp)
movq    -16(%rbp), %rax
movl    $15, (%rax)
movq    -8(%rbp), %rax
movl    $16, (%rax)
movq    -16(%rbp), %rax
movl    (%rax), %edx
movq    -8(%rbp), %rax
movl    (%rax), %eax
addl    %edx, %eax
movl    %eax, %esi
leaq    _ZSt4cout(%rip), %rdi
call    _ZNSolsEi@PLT
movq    -16(%rbp), %rax
testq   %rax, %rax
je      .L2
```

The addresses of the variables f and s are getting stored at -8(rbp), and -16(rbp) respectively.

```
(gdb) nexti 4
0x00000000004007ec in main ()
(gdb) x $rbp - 0x10
0x7fffffffde80: 0x00613c20
(gdb) nexti 3
0x00000000004007fa in main ()
(gdb) x $rbp - 0x8
0x7fffffffde88: 0x00613c40
```

As the addresses of the two heap dynamic variables is different, hence they are allocated different dynamic memories at different places. Hence, the variables

stored in the heap dynamic variables are allocated at non consecutive locations of memory which is totally different in the case of a stack .
But if we see the storage of the variables in the memory, and this is done at consecutive locations of memory as -8(%RBP) , -16(%RBP) , etc.

Q5.
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 programs in C / C++ define the specification of the language elements, but the method of implementation of the switch statement is free upon the choice of the vendor . As discussed in the book, there are mainly three ways of implementing switch case statements and emulation in programming languages.

The first one is for a combination of comparisons such as BNE , which means that if the current value matches one of the switch cases , the execution will be headed over to that.

The second method will be loading the whole function pointers into the table , further then using that variable to add to the base address of the table , in order to select the function pointer that we need to jump upon.

With reference to the book, the case as well as the label values of the conditional branches are stored in the form of jump table
A very inefficient technique that is generally used by the compilers is linear search upon the jump table.
Also, in addition to that, the hash table is also built on the basis of the segment tables, which resulted in approximately equal times to choose from the selectable segments.

Another method, is using array whose indices are the case values and their values built up on the basis of segment labels.

```
        cmpl    $1, -4(%rbp)
        je      .L6
        jmp     .L3
.L5:
        leaq    .LC0(%rip), %rdi
        movl    $0, %eax
        call    printf@PLT
        jmp     .L7
.L6:
        leaq    .LC1(%rip), %rdi
        movl    $0, %eax
        call    printf@PLT
        jmp     .L7
.L4:
        leaq    .LC2(%rip), %rdi
        movl    $0, %eax
        call    printf@PLT
        jmp     .L7
.L2:
        leaq    .LC3(%rip), %rdi
        movl    $0, %eax
        call    printf@PLT
        jmp     .L7
.L3:
        leaq    .LC4(%rip), %rdi
        movl    $0, %eax
        call    printf@PLT
        nop
.L7:
        movl    $0, %eax
        leave
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   main, .-main
```

```
.L5:
        leaq   .LC0(%rip), %rdi
        movl   $0, %eax
        call   printf@PLT
        jmp    .L7
```

So for every called L3,L4,L5…..
The following statement compares the given choice value if that is equal to the given value of choice right now, it will jump to that corresponding function if equal else not.

There are only three cases that i have mentioned in the switch case function, now for the given assembly code the

```
        .text
        .section        .rodata
.LC0:
        .string "Case 0"
.LC1:
        .string "Case 1"
.LC2:
        .string "Case 2"
.LC3:
        .string "Case 3"
.LC4:
        .string "Default Case"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        endbr64
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
```

.LC0 gets called when the choice is 0
.LC1 gets called when the choice is 1
.LC2 gets called when the choice is 2
.LC3 gets called when the choice is 3
.LC4 gets called when there is no choice matched after the linear traversal

```
        .string "Default Case"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        endbr64
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        subq    $16, %rsp
        movl    $2, -4(%rbp)
        cmpl    $3, -4(%rbp)
        je      .L2
        cmpl    $3, -4(%rbp)
        jg      .L3
        cmpl    $2, -4(%rbp)
        je      .L4
        cmpl    $2, -4(%rbp)
        jg      .L3
        cmpl    $0, -4(%rbp)
        je      .L5
        cmpl    $1, -4(%rbp)
        je      .L6
        jmp     .L3
.L5:
        leaq    .LC0(%rip), %rdi
        movl    $0, %eax
        call    printf@PLT
```

This is a sequential search method of linear sequential search (ALSO CALLED AS ONE LEVEL JUMP TABLE), though this could be followed up in the lookup table and using binary search also.

Now, when the case values are far apart , the switch case implementation uses the concept of binary search .

The following is the assembly code of the main function , which compares the corresponding values 100, 200, 300 and soo on till 1000 with the default case only. The corresponding assembly code is given as, for the above code :

```c
int main()
{
	int x =200;

	switch(x)
	{
		case 100:
			{
			printf("Case 100");
			break;
			}
		case 200:
			{
			printf("Case 200");
			break;
			}
		case 300:
			{
			printf("Case 300");
			break;
			}
		case 400:
			{
			printf("Case 400");
			break;
			}
		case 500:
			{
			printf("Case 500");
			break;
			}
		case 600:
			{
			printf("Case 600");
			break;
			}
		case 700:
			{
			printf("Case 700");
			break;
			}
		case 800:
			{
```

```
main:
.LFB0:
	.cfi_startproc
	endbr64
	pushq	%rbp
	.cfi_def_cfa_offset 16
	.cfi_offset 6, -16
	movq	%rsp, %rbp
	.cfi_def_cfa_register 6
	subq	$16, %rsp
	movl	$2, -4(%rbp)
	cmpl	$1000, -4(%rbp)
	je	.L2
	cmpl	$1000, -4(%rbp)
	jg	.L3
	cmpl	$900, -4(%rbp)
	je	.L4
	cmpl	$900, -4(%rbp)
	jg	.L3
	cmpl	$800, -4(%rbp)
	je	.L5
	cmpl	$800, -4(%rbp)
	jg	.L3
	cmpl	$700, -4(%rbp)
	je	.L6
	cmpl	$700, -4(%rbp)
	jg	.L3
	cmpl	$600, -4(%rbp)
	je	.L7
	cmpl	$600, -4(%rbp)
	jg	.L3
	cmpl	$500, -4(%rbp)
	je	.L8
	cmpl	$500, -4(%rbp)
	jg	.L3
	cmpl	$400, -4(%rbp)
	je	.L9
	cmpl	$400, -4(%rbp)
	jg	.L3
```

```
	.section	.rodata
.LC0:
	.string "Case 100"
.LC1:
	.string "Case 200"
.LC2:
	.string "Case 300"
.LC3:
	.string "Case 400"
.LC4:
	.string "Case 500"
.LC5:
	.string "Case 600"
.LC6:
	.string "Case 700"
.LC7:
	.string "Case 800"
.LC8:
	.string "Case 900"
.LC9:
	.string "Case 1000"
.LC10:
	.string "Default Case"
	.text
	.globl	main
	.type	main, @function
```

The above are the cases of the different case statements, since the values are far apart , hence instead of the single sequential approach , it uses the concept of binary search.

```
	jmp	.L13
.L12:
	leaq	.LC1(%rip), %rdi
	movl	$0, %eax
	call	printf@PLT
	jmp	.L13
.L10:
	leaq	.LC2(%rip), %rdi
	movl	$0, %eax
	call	printf@PLT
	jmp	.L13
.L9:
	leaq	.LC3(%rip), %rdi
	movl	$0, %eax
```

```
.L11:
        leaq    .LC0(%rip), %rdi
        movl    $0, %eax
        call    printf@PLT
        jmp     .L13
.L12:
        leaq    .LC1(%rip), %rdi
        movl    $0, %eax
        call    printf@PLT
        jmp     .L13
.L10:
        leaq    .LC2(%rip), %rdi
        movl    $0, %eax
        call    printf@PLT
        jmp     .L13
.L9:
        leaq    .LC3(%rip), %rdi
        movl    $0, %eax
        call    printf@PLT
        jmp     .L13
.L8:
        leaq    .LC4(%rip), %rdi
        movl    $0, %eax
        call    printf@PLT
        jmp     .L13
.L7:
        leaq    .LC5(%rip), %rdi
        movl    $0, %eax
        call    printf@PLT
        jmp     .L13
.L6:
        leaq    .LC6(%rip), %rdi
        movl    $0, %eax
        call    printf@PLT
        jmp     .L13
.L5:
        leaq    .LC7(%rip), %rdi
        movl    $0, %eax
        call    printf@PLT
        jmp     .L13
```

Then when the value matches, it finally results in the choice made , now since the difference between the values is quite large , it will mean use the approach of binary search instead of the one level lookup table to search for the correct corresponding value .

By generating the assembly listing, we take a look from bottom up by checking the case operations first. Also, the hybrid of both the one level lookup table as well as the binary search is implemented in programs involving C/C++. Only difference is that the labels are renumbered, on the basis of binary search.

The translated assembly code only reflects the binary search.

```
switch(x)
{
        case 10000:
                {
                printf("Case 10000");
                break;
                }
        case 1200:
                {
                printf("Case 1200");
                break;
                }
        case 300:
                {
                printf("Case 300");
                break;
                }
        case 4000:
                {
                printf("Case 4000");
                break;
                }
        case 5100:
                {
                printf("Case 5100");
                break;
                }
        case 60:
                {
                printf("Case 60");
                break;
                }
        case 7:
                {
                printf("Case 7");
                break;
                }
        case 8000:
                {
                printf("Case 8000");
                break;
                }
```

Even when the values are randomly aligned, the same values are seen to be observed.

```
	.cfi_def_cfa_register 6
	subq	$16, %rsp
	movl	$2000, -4(%rbp)
	cmpl	$10000, -4(%rbp)
	je	.L2
	cmpl	$10000, -4(%rbp)
	jg	.L3
	cmpl	$8000, -4(%rbp)
	je	.L4
	cmpl	$8000, -4(%rbp)
	jg	.L3
	cmpl	$5100, -4(%rbp)
	je	.L5
	cmpl	$5100, -4(%rbp)
	jg	.L3
	cmpl	$4000, -4(%rbp)
	je	.L6
	cmpl	$4000, -4(%rbp)
	jg	.L3
	cmpl	$1200, -4(%rbp)
	je	.L7
	cmpl	$1200, -4(%rbp)
	jg	.L3
	cmpl	$1000, -4(%rbp)
	je	.L8
	cmpl	$1000, -4(%rbp)
	jg	.L3
	cmpl	$300, -4(%rbp)
	je	.L9
	cmpl	$300, -4(%rbp)
	jg	.L3
	cmpl	$60, -4(%rbp)
	je	.L10
	cmpl	$60, -4(%rbp)
	jg	.L3
	cmpl	$7, -4(%rbp)
	je	.L11
	cmpl	$9, -4(%rbp)
	je	.L12
	jmp	.L3
```

```
	je	.L12
	jmp	.L3
.L2:
	leaq	.LC0(%rip), %rdi
	movl	$0, %eax
	call	printf@PLT
	jmp	.L13
.L7:
	leaq	.LC1(%rip), %rdi
	movl	$0, %eax
	call	printf@PLT
	jmp	.L13
.L9:
	leaq	.LC2(%rip), %rdi
	movl	$0, %eax
	call	printf@PLT
	jmp	.L13
.L6:
	leaq	.LC3(%rip), %rdi
	movl	$0, %eax
	call	printf@PLT
	jmp	.L13
.L5:
	leaq	.LC4(%rip), %rdi
	movl	$0, %eax
	call	printf@PLT
	jmp	.L13
.L10:
	leaq	.LC5(%rip), %rdi
	movl	$0, %eax
	call	printf@PLT
	jmp	.L13
.L11:
	leaq	.LC6(%rip), %rdi
```

Q6.

Comment on how C/C++ compiler uses stack to implement a recursive program.

Writing a program (recursive),

```
#include<stdio.h>

int sum(int n)
{
        if(n<=0)
        return 0;

        return n+ sum(n-1);
}
int main()
{
        int n=25;
        int s = sum(n);
        printf("%d", s);
}
```

The above is the recursive program for finding the sum of first n numbers where n=25,
Which will recursively calculate the sum of the first n numbers .
The assembly code of the above program will be written as:

```
        .file   "Q5.c"
        .text
        .globl  sum
        .type   sum, @function
sum:
.LFB0:
        .cfi_startproc
        endbr64
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        subq    $16, %rsp
        movl    %edi, -4(%rbp)
        cmpl    $0, -4(%rbp)
        jg      .L2
        movl    $0, %eax
        jmp     .L3
.L2:
        movl    -4(%rbp), %eax
        subl    $1, %eax
        movl    %eax, %edi
        call    sum
        movl    -4(%rbp), %edx
        addl    %edx, %eax
.L3:
        leave
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   sum, .-sum
        .section        .rodata
.LC0:
        .string "%d"
        .text
        .globl  main
        .type   main, @function
main:
.LFB1:
```

```
main:
.LFB1:
        .cfi_startproc
        endbr64
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        subq    $16, %rsp
        movl    $25, -8(%rbp)
        movl    -8(%rbp), %eax
        movl    %eax, %edi
        call    sum
        movl    %eax, -4(%rbp)
        movl    -4(%rbp), %eax
        movl    %eax, %esi
        leaq    .LC0(%rip), %rdi
        movl    $0, %eax
        call    printf@PLT
        movl    $0, %eax
        leave
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE1:
        .size   main, .-main
```

In the assembly code of the main program, we could see that where the actual parameter passed has a value equal to 25 , and hence which passed to the function "sum" via EDI.

Also, the formal parameters which are stored as local variables, are recursively called until the value of EDI is equal to zero. As we could observe that the value is decreasing every time by 1 and the function "sum" is again called when the decreased value is passed as a parameter in the function "sum".

```
        jmp     .L3
.L2:
        movl    -4(%rbp), %eax
        subl    $1, %eax
        movl    %eax, %edi
        call    sum
        movl    -4(%rbp), %edx
        addl    %edx, %eax
.L3:
        leave
        .cfi_def_cfa 7, 8
        ret
```

So, when the value of EDI becomes zero, and again the backtracking happens, then the final sum in eax gets updated.

In C/C++, the compiler uses stack in order to return the address of the functions, also, the structures are normally allocated to stack.
For the inbuilt behaviour of the function call, it occupies a memory in the stack to store the execution of the program. When the function ends, it returns to its calling statements and deallocated from the stack.