Indian Institute of Information Technology Allahabad

PPL Assignment - C3 (May 2021)

Fourth semester B.Tech (IT) - All Sections

Name : Aditya Aggarwal Roll Number: IIT2019210

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. Note: refer chapter 9 and 10 of the book.

Solution)

Example C Code:

```c
#include<stdio.h>
int calculate(int p, int q, int r, int s, int t){
        int result; result=p*q-r*s-t; return result;
}
int main(){
        int p,q,r,s,t; p=9; q=12; r=3; s=4; t=4;
        int result=calculate(p,q,r,s,t); printf("%d\n",result);
        return 0;
}
```

Assembly Version using gcc –S command:

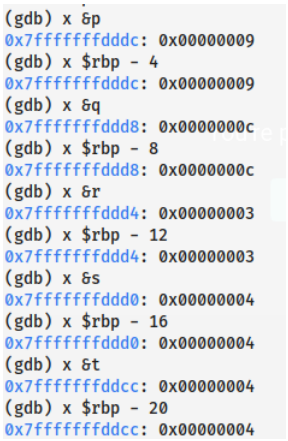
main:

```
pushq   %rbp
movq    %rsp, %rbp
subq    $32, %rsp
movl    $9, -4(%rbp)
movl    $12, -8(%rbp)
movl    $3, -12(%rbp)
movl    $4, -16(%rbp)
movl    $4, -20(%rbp)
movl    -20(%rbp), %edi
movl    -16(%rbp), %ecx
movl    -12(%rbp), %edx
movl    -8(%rbp), %esi
movl    -4(%rbp), %eax
movl    %edi, %r8d
movl    %eax, %edi
call    calculate
movl    %eax, -24(%rbp)
movl    -24(%rbp), %eax
movl    %eax, %esi
leaq    .LC0(%rip), %rdi
movl    $0, %eax
call    printf@PLT
movl    $0, %eax
leave
ret
```

calculate:

```
pushq   %rbp
movq    %rsp, %rbp
movl    %edi, -20(%rbp)
movl    %esi, -24(%rbp)
movl    %edx, -28(%rbp)
movl    %ecx, -32(%rbp)
movl    %r8d, -36(%rbp)
movl    -20(%rbp), %eax
imull   -24(%rbp), %eax
movl    %eax, %edx
movl    -28(%rbp), %eax
imull   -32(%rbp), %eax
movl    %eax, %ecx
movl    %edx, %eax
subl    %ecx, %eax
subl    -36(%rbp), %eax
movl    %eax, -4(%rbp)
movl    -4(%rbp), %eax
popq    %rbp
ret
```

rbp - Points the base of current stack frame.
rsp - Points the top of current stack frame.



First of all, the base address is stored using instruction pushq %rbp, then value of rsp is copied to rbp by using instruction movq %rsp, $rbp, and 32 bytes space is allocated to stack for the function local variables using instruction subq $32, %rsp .

The actual parameters are stored into rbp registers by incrementing position by 4 bytes each. For example first parameter is stored at -4($rbp), then next is stored 4 bytes ahead of previous that is -8($rbp). The screenshot attached will tell that how the value is stored using gdb.
x &p will tell value of parameter p and x $rbp - 4 will tell value at address rbp-4. And similarly for other 4 variables q,r,s,t.
For this we need to compile using gcc -g -O0 q1.c
Since, the values at &p and $rbp-4 is same, we can say that the parameter is stored here and similarly for others.

Further their values are saved into edi, ecx, edx, esi, eax registers that is done by copying the values using rbp. Example : movl -20(%rbp), %edi copies value of variable to edi register.

eax register is used to store the return value of a function but at this moment , one variable's value is stored into eax so first this variable value needs to stored somewhere else , that is it would be stored in edi register, further we need to save value of edi register somewhere else , that is the r8d register. So, commands movl %edi, %r8d and movl %eax, %edi copies edi to e8d first and then copies eax to edi. At this moment, the actual paramters are transferred into callee-save registers.
The 5 parameters that are passed into calculate , their values are stored into edi(stores p), esi(stores q), edx(stores r), ecx(stores s) and r8d(stores t).

After it, the 'calculate' function is called. For that the eip is pushed(by calculate) to store the return address of the 'main' function. This address is stored in 'calculate' function's stack and using jump instruction, jump is done to 'calculate'. At this moment, caller(main function) copies return address in the stack space of called function(calculate).

Now in function 'calculate' , rbp(base pointer) is saved using pushq $rbp and then the rsp is copied to rbp using instruction: movq %rsp, %rbp.
At this moment, the values of p,q,r,s,t from caller(main) function are stored in the called(calculate) function. This values are stored in its local variables. The values are copied using movl %edi, -20(%rbp) , movl %esi, -24(%rbp) , movl %edx, -28(%rbp) , movl %ecx, -32(%rbp) , movl %r8d, -36(%rbp).

Finally, the calculations are done the result is copies to eax register using movl -4(%rbp), %eax.

Q2. 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

Solution)

C Code with variable ‘t’ send using pointer.

```
#include<stdio.h>
int calculate(int p, int q, int r, int s, int *t){
        int result;     result=p*q-r*s-(*t);    return result;
}
int main(){
        int p,q,r,s,t;    p=9;    q=12;   r=3;    s=4;    t=4;
        int result=calculate(p,q,r,s,&t); printf("%d\n",result);
        return 0;
}
```

Assembly Version of C Code:

main:

```
pushq   %rbp
movq    %rsp, %rbp
subq    $32, %rsp
movl    $9, -4(%rbp)
movl    $12, -8(%rbp)
movl    $3, -12(%rbp)
movl    $4, -16(%rbp)
movl    $4, -24(%rbp)
leaq    -24(%rbp), %rdi
movl    -16(%rbp), %ecx
movl    -12(%rbp), %edx
movl    -8(%rbp), %esi
movl    -4(%rbp), %eax
movq    %rdi, %r8
movl    %eax, %edi
call    calculate
movl    %eax, -20(%rbp)
movl    -20(%rbp), %eax
movl    %eax, %esi
leaq    .LC0(%rip), %rdi
movl    $0, %eax
call    printf@PLT
movl    $0, %eax
leave
ret
```

calculate:

```
pushq   %rbp
movq    %rsp, %rbp
movl    %edi, -20(%rbp)
movl    %esi, -24(%rbp)
movl    %edx, -28(%rbp)
movl    %ecx, -32(%rbp)
movq    %r8, -40(%rbp)
movl    -20(%rbp), %eax
imull   -24(%rbp), %eax
movl    %eax, %edx
movl    -28(%rbp), %eax
imull   -32(%rbp), %eax
subl    %eax, %edx
movq    -40(%rbp), %rax
movl    (%rax), %ecx
movl    %edx, %eax
subl    %ecx, %eax
movl    %eax, -4(%rbp)
movl    -4(%rbp), %eax
popq    %rbp
ret
```

C++ Code with variable ‘t’ send using reference variable.

```
#include<bits/stdc++.h>
using namespace std;
int calculate(int p, int q, int r, int s, int& t){
        int result;     result=p*q-r*s-t;       return result;
}
int main(){
        int p,q,r,s,t;    p=9;    q=12;   r=3;    s=4;    t=4;
        int result=calculate(p,q,r,s,t);
        cout<<result<<endl;
        return 0;
}
```

Assembly Version of C++ Code:

```
main:
        pushq   %rbp
        movq    %rsp, %rbp
        subq    $32, %rsp
        movl    $9, -4(%rbp)
        movl    $12, -8(%rbp)
        movl    $3, -12(%rbp)
        movl    $4, -16(%rbp)
        movl    $4, -24(%rbp)
        leaq    -24(%rbp), %rdi
        movl    -16(%rbp), %ecx
        movl    -12(%rbp), %edx
        movl    -8(%rbp), %esi
        movl    -4(%rbp), %eax
        movq    %rdi, %r8
        movl    %eax, %edi
        call    _Z9calculateiiiiRi
        movl    %eax, -20(%rbp)
        movl    -20(%rbp), %eax
        movl    %eax, %esi
        leaq    _ZSt4cout(%rip), %rdi
        call    _ZNSolsEi@PLT
        movq    %rax, %rdx
        movq    _ZSt4endlIcSt11char_traitsI-
cEERSt13basic_ostreamIT_T0_ES6_@GOTP-
CREL(%rip), %rax
        movq    %rax, %rsi
        movq    %rdx, %rdi
        call    _ZNSolsEPFRSoS_E@PLT
        movl    $0, %eax
        leave
        ret
```

```
_Z9calculateiiiiRi:
.LFB8378:
        pushq   %rbp
        movq    %rsp, %rbp
        movl    %edi, -20(%rbp)
        movl    %esi, -24(%rbp)
        movl    %edx, -28(%rbp)
        movl    %ecx, -32(%rbp)
        movq    %r8, -40(%rbp)
        movl    -20(%rbp), %eax
        imull   -24(%rbp), %eax
        movl    %eax, %edx
        movl    -28(%rbp), %eax
        imull   -32(%rbp), %eax
        subl    %eax, %edx
        movq    -40(%rbp), %rax
        movl    (%rax), %ecx
        movl    %edx, %eax
        subl    %ecx, %eax
        movl    %eax, -4(%rbp)
        movl    -4(%rbp), %eax
        popq    %rbp
        ret
```

We can observe that the assembly version of c and cpp are same inspite of the fact that I used pointer for passing ‘t’ in c and reference variable for passing ‘t’ in cpp. So, now it means the answer to the question asked is same for both of them.
Like in q1, the values are copied to rbp register in the same manner for first four variables and variable ‘t’ is stored at rbp-24 instead of rbp-20 as in q1 because of the stack buffer overflow. So, the canary value is stored in the stack to check the overflow.
The second difference observed is that before the ‘calculate’ function calling, instead of copying the value of ‘t’ using movl , it is copying its address using instruction leaq -24(%rbp), %rdi.
The reason behind this is that in q1 , paramter was passed as a value so if its value changes in function, there would no effect on original parameter but in this case, parameter is passed as a pointer/ reference variable so if its value changes in function, the same impact must be on the original variable so instead of value, its address is passed.
The third change is that in this the address of ‘t’ is stored in edi register whereas in q1 its value was stored in rdi register.
The fourth change is that since rdi is used in leaq .LCO(%rip), %rdi instruction too. So, its value is copied to r8 register before ‘calculate’ call.
The last change observed is that in ‘calculate’ function the value is acessed using movq instruction instead of movl instruction. The reason is that movl is used to move 32 bits and movq is used to move 64 bits. Here we need to store the address and not the value.

Finally, the return address is stored in the similar way. The first four parameters(p,q,r and s) are passed in the similar manner but fifth paramater ‘t’’s address is passed instead of value.

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

Sol) To analyze the fixed stack dynamic and stack dynamic arrays, I have created 2 C programs which calculates the sum of 50 garbage values.

Fixed stack dynamic C Program :

```c
#include<stdio.h>
int create_fixed_stack_dynamic_array(){
        int my_fixed_stack_dynamic_array[50];
        int rand_sum=0;
        //Calculates the sum of 50 garbage values
        for(int i=0;i<50;i++){
                rand_sum+=my_fixed_stack_dynamic_array[i];
        }
        return rand_sum;
}
int main(){
        printf("%d",create_fixed_stack_dynamic_array());
        return 0;
}
```

Assembly Version:

```
main:
        pushq   %rbp
        movq    %rsp, %rbp
        movl    $0, %eax
        call    create_fixed_stack_dynamic_array
        movl    %eax, %esi
        leaq    .LC0(%rip), %rdi
        movl    $0, %eax
        call    printf@PLT
        movl    $0, %eax
        popq    %rbp
        ret
```

```
create_fixed_stack_dynamic_array:
        pushq   %rbp
        movq    %rsp, %rbp
        subq    $88, %rsp
        movl    $0, -4(%rbp)
        movl    $0, -8(%rbp)
        jmp     .L2
```

```
.L2:
        cmpl    $49, -8(%rbp)
        jle     .L3
        movl    -4(%rbp), %eax
        leave
        ret
```

```
.L3:
        movl    -8(%rbp), %eax
        cltq
        movl    -208(%rbp,%rax,4), %eax
        addl    %eax, -4(%rbp)
        addl    $1, -8(%rbp)
```

It can be observed that the variable my_fixed_stack_dynamic_array has been allocated the fixed stack dynamically. This means that size of my_fixed_stack_dynamic_array will be known at compile time, but the its memory will be allocateed at run time.
Now, we can observe that in the assembly version of this code, before calling the function ‘create_fixed_stack_dynamic_array’ from main, there is no line of code which tells us about the array allocation. Since, its size is fixed, and known already so, only memory allocation has been done.

Stack dynamic C Program :

```c
#include<stdio.h>
int create_stack_dynamic_array(int array_size){
        int my_stack_dynamic_array[array_size];
        int rand_sum=0;
        //Calculates the sum of 50 garbage values
        for(int i=0;i<array_size;i++){
                rand_sum+=my_stack_dynamic_array[i];
        }
        return rand_sum;
}
int main(){
        printf("%d",create_stack_dynamic_array(50));
        return 0;
}
```

Assembly Version:

```
main:
        pushq   %rbp
        movq    %rsp, %rbp
        movl    $50, %edi
        call    create_stack_dynamic_array
        movl    %eax, %esi
        leaq    .LC0(%rip), %rdi
        movl    $0, %eax
        call    printf@PLT
        movl    $0, %eax
        popq    %rbp
        ret
```

```
create_stack_dynamic_array:
        pushq   %rbp
        movq    %rsp, %rbp
        subq    $48, %rsp
        movl    %edi, -36(%rbp)
        movq    %rsp, %rax
        movq    %rax, %rcx
        movl    -36(%rbp), %eax
        movslq  %eax, %rdx
        subq    $1, %rdx
        movq    %rdx, -16(%rbp)
        movslq  %eax, %rdx
        movq    %rdx, %r10
        movl    $0, %r11d
        movslq  %eax, %rdx
        movq    %rdx, %r8
        movl    $0, %r9d
        cltq
        leaq    0(,%rax,4), %rdx
        movl    $16, %eax
        subq    $1, %rax
        addq    %rdx, %rax
        movl    $16, %esi
        movl    $0, %edx
        divq    %rsi
        imulq   $16, %rax, %rax
        subq    %rax, %rsp
        movq    %rsp, %rax
        addq    $3, %rax
        shrq    $2, %rax
        salq    $2, %rax
        movq    %rax, -24(%rbp)
        movl    $0, -8(%rbp)
        movl    $0, -4(%rbp)
        jmp     .L2
```

```
.L3:
        movq    -24(%rbp), %rax
        movl    -4(%rbp), %edx
        movslq  %edx, %rdx
        movl    (%rax,%rdx,4), %eax
        addl    %eax, -8(%rbp)
        addl    $1, -4(%rbp)
.L2:
        movl    -4(%rbp), %eax
        cmpl    -36(%rbp), %eax
        jl      .L3
        movl    -8(%rbp), %eax
        movq    %rcx, %rsp
        leave
        ret
```

It can be observed that the variable my_stack_dynamic_array has been allocated the stack dynamically. This means that size of my_stack_dynamic_array will be unknown at compile time, and will be only known at the run time. Memory will be allocateed at run time that is similar to previous case.
Now, we can observe that in the assembly version of this code, before calling the function ‘create_stack_dynamic_array’ from main, the paramter ’array_size’ that is passed in function, it value is stored in edi register using instruction movl $50, %edi. That is the size of array that is passed into the function.
In the function my_stack_dynamic_array, the value passed (i.e. size of array in edi register) is stored locally using instruction movl %edi, -36(%rbp).

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.

Sol) For this , the example cpp program is attached in which 2 heap dynamic variables, and further the factorial is calculated for their sum i.e. factorial(heap_var1 + heap_var2)

Cpp Code:

```
#include<bits/stdc++.h>
using namespace std;
int factorial(int n){
        if(n<=1) return 1;
        return (n*factorial(n-1));
}
int main(){
        int* heap_dynamic_1=new int;
        int* heap_dynamic_2=new int;
        *heap_dynamic_1=3;              *heap_dynamic_2=2;
        cout<<factorial(*heap_dynamic_1 + *heap_dynamic_2);
        return 0;
}
```

Assembly Version:

```
main:
        pushq   %rbp
        movq    %rsp, %rbp
        subq    $16, %rsp
        movl    $4, %edi
        call    _Znwm@PLT
        movq    %rax, -8(%rbp)
        movl    $4, %edi
        call    _Znwm@PLT
        movq    %rax, -16(%rbp)
        movq    -8(%rbp), %rax
        movl    $3, (%rax)
        movq    -16(%rbp), %rax
        movl    $2, (%rax)
        movq    -8(%rbp), %rax
        movl    (%rax), %edx
        movq    -16(%rbp), %rax
        movl    (%rax), %eax
        addl    %edx, %eax
        movl    %eax, %edi
        call    _Z9factoriali
        movl    %eax, %esi
        leaq    _ZSt4cout(%rip), %rdi
        call    _ZNSolsEi@PLT
        movl    $0, %eax
        ret
```

```
0x00005555555551b4 <+36>:    mov    -0x8(%rbp),%rax
0x00005555555551b8 <+40>:    movl   $0x3,(%rax)
0x00005555555551be <+46>:    mov    -0x10(%rbp),%rax
0x00005555555551c2 <+50>:    movl   $0x2,(%rax)
```

While debugging the code , we can observe that the value of heap variables (heap_dynamic1 and heap_dynamic2) are stored in rbp-0x8 and rbp-0x10 respectively which can be seen in above diagram.

In the below diagram, I have confirmed that the variables are stored in rbp-0x8 and rbp-0x10 respectively.

```
(gdb) x &heap_dynamic_1
0x7fffffffddc8: 0x5556aeb0
(gdb) x $rbp-0x8
0x7fffffffddc8: 0x5556aeb0
(gdb) x &heap_dynamic_2
0x7fffffffddc0: 0x5556aed0
(gdb) x $rbp-0x10
0x7fffffffddc0: 0x5556aed0
```

Since heap allocates the memory randomly, so variables are not stored in consecutive address of memory.
In stack dynamic variables, like in previous question, they were stored in the difference of 4 as rbp-4, rbp-8, rbp-12, rbp-16, rbp-20.....(Reason behind this is that the memory is allocated linearly in stack in a sequencial order). Rather in this case, the first variable heap_dynamic1 is stored at rbp-0x8 and second variable heap_dynamic2 is stored at rbp-0x10 which are completely random. In this way, the heap dynamic variable is different from the stack dynamic variables.
Comparing the heap dynamic variables and static variables, the data segment is the place where static variables are stored which is not the case with heap dynamic variables.