Alex Yang
ahy3nz

```
#include <iostream>
#include <cstdlib>
#include <stdio.h>
#include <stdlib.h>
#include <string>

using namespace std;

class testclass {
public:
  int n;
  char c;
   testclass(int stuff);
  int getNum();
  int getPrivNum();
  char getPrivChar();
  char getChar();
 private:
  int privn;
  char privc;

};

int main () {
  testclass stuff = testclass(3);
  cout<<stuff.getNum()<<endl;
  cout<<stuff.getChar()<<endl;
  cout<<stuff.getPrivNum()<<endl;
  cout<<stuff.getPrivChar()<<endl;
          return 0;
}

testclass::testclass(int stuff) {
  n = stuff;
  c = (char)stuff;
  privn = stuff;
  privc = (char)stuff;
}

int testclass::getNum() {
  return n;
```

```
}

int testclass::getPrivNum() {
  return privn;
}

char testclass::getPrivChar() {
  return privc;
}

char testclass::getChar() {
  return c;
}

.cfi_startproc
        push    rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        mov     rbp, rsp
        .cfi_def_cfa_register 6
        sub     rsp, 32
        lea     rax, [rbp-32]
        mov     esi, 3
        mov     rdi, rax
        call    _ZN9testclassC1Ei
```

This is a series of standard prologues and allocating space (and padding). The value of the parameter 3, is stored in esi, and rax and rdi holds the value at memory specified in [rbp - 32]. Then the call to the testclass instructor is made

```
_ZN9.testclassC2Ei:
.LFB972:
        .cfi_startproc
        push    rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        mov     rbp, rsp
        .cfi_def_cfa_register 6
        mov     QWORD PTR [rbp-8], rdi
        mov     DWORD PTR [rbp-12], esi
        mov     rax, QWORD PTR [rbp-8]
        mov     edx, DWORD PTR [rbp-12]
        mov     DWORD PTR [rax], edx
```

```
mov     eax, DWORD PTR [rbp-12]
mov     edx, eax
mov     rax, QWORD PTR [rbp-8]
mov     BYTE PTR [rax+4], dl
mov     rax, QWORD PTR [rbp-8]
mov     edx, DWORD PTR [rbp-12]
mov     DWORD PTR [rax+8], edx
mov     eax, DWORD PTR [rbp-12]
mov     edx, eax
mov     rax, QWORD PTR [rbp-8]
mov     BYTE PTR [rax+12], dl
pop     rbp
.cfi_def_cfa 7, 8
ret
.cfi_endproc
```

The value of rdi holds the integer “stuff.” rbp-8, a memory location on the stack, holds an address, so the QWORD PTR [rbp-8] will store value in rdi in the address specified by rbp-8, which is an address for a variable in testclass. Similarly esi, it is stored in the address specified by rbp-12. This means, we’re pushing addresses to testclass’s variables on the stack. When we have to modify these variables, we’re loading these addresses from the stack, which are offsets from the stack pointers. The value stored at the address specified by rbp-8 is another address, which is moved into register eax. This means eax holds an address. Further operations show that variables in a class are listed as addresses on a stack.To modify and store values in these variables, we have to access these addresses and change the values at the addresses specified by the memory locations on the stack/activation record.

```
lea     rax, [rbp-32]
mov     rdi, rax
call    _ZN9testclass6getNumEv
```

This code is located in the main method. The address at rbp-32 is loaded into rax. The series of memory at rbp-32 refers to the variables in testclass.

```
 push   rbp
.cfi_def_cfa_offset 16
.cfi_offset 6, -16
mov     rbp, rsp
.cfi_def_cfa_register 6
mov     QWORD PTR [rbp-8], rdi
mov     rax, QWORD PTR [rbp-8]
mov     eax, DWORD PTR [rax]
pop     rbp
.cfi_def_cfa 7, 8
```

```
ret
.cfi_endproc
```

In this get subroutine, we are accessing a public variable in testclass. Rdi holds an address and is stored into an offset of the basepointer and into rax. By accessing the value specified at memory rax, we obtain the value for the variable “n” in test class. In get subroutines, the parameters we pass in are addresses in the stack. We access the address to access the variables stored in testclass. These variables are located in memory spots elsewhere on the stack, so we need to remember the addresses of these variables.

Outside the function, variables are stored on the stack in some series of memory locations. We can perform offsets from the stack pointer in the main routine to access the addresses of the variables. Within the subroutine, we can still access the data based on offsets from the stack pointer since we store addresses to the local variables on the stack in offsets from that base pointer that hold addresses to the local variables.