Name: Tho Nguyen
Email ID: tnn7yc
File name: postlab8.pdf
Date: 10/22/18

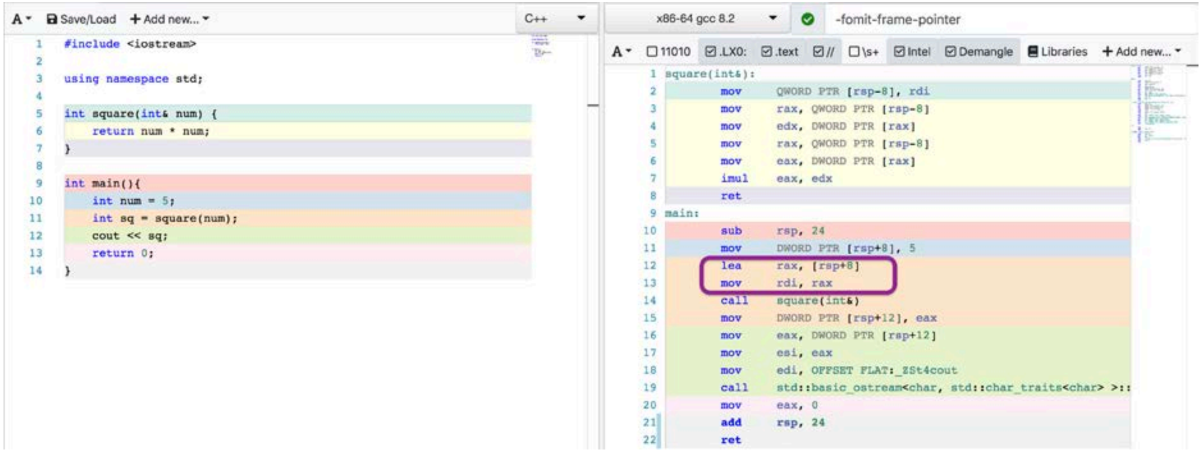
Parameters Passing

1. How are variables (ints, chars, pointers, floats, etc.) passed by value? How are they passed by reference?

At the beginning of a program, the complier makes space for variables in the stack by subtracting rsp. Then the compiler save the values of those variables onto the stack according to their sizes (4 bytes for int, 8 bytes for long, etc.)

When variables are passed by value, before a callee is called, the caller moves copy of values of parameters into appropriate registers. The registers used to have parameters moved into are determined based on the size of the parameters. For example, for int parameter, the complier uses edi, esi, etc. since int parameter takes up only the lower 4 bytes of a register. On the other hand, when passing parameter of longer size, such as long, the compiler passes a copy of value of parameter into rdi, rsi, etc.

When variables are passed by reference, before the caller calls a callee, the values that being moved to the parameter registers are not copy of value of parameter, but pointer to memory address that holds the original values themselves. For example in Screenshoot 3, the value of rdi is set, before calling the callee, to the memory address that holds num. Then inside the callee, rdi is indirectly dereferenced, its value is copied into return value register and operation happens on the return value register. When callee is done and the return value is returned to the caller, the caller set value inside the memory address that holds the original parameter to the return value.



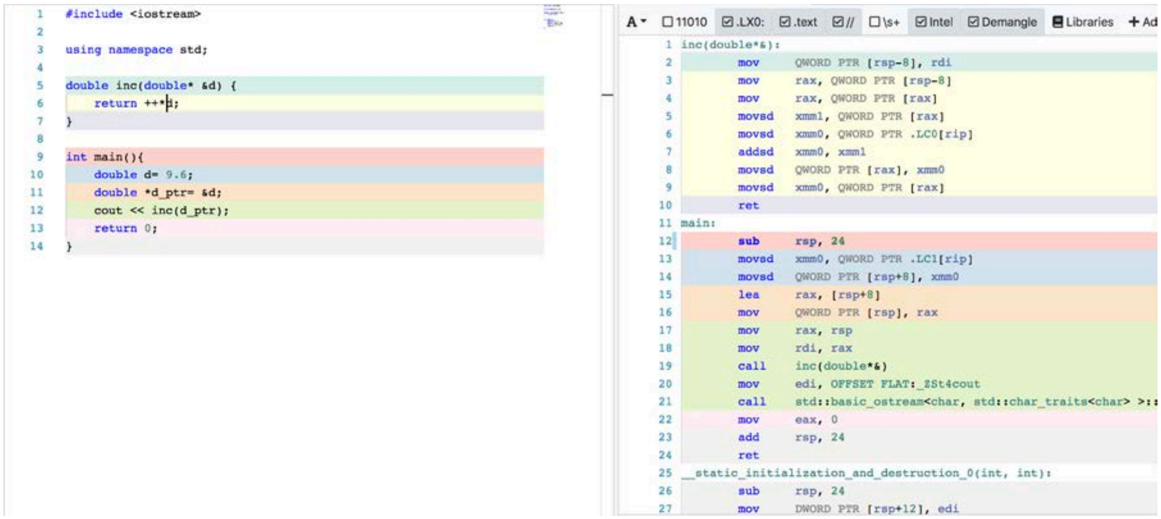
x86-64 gcc 8.2
-fomit-frame-pointer

```
#include <iostream>

using namespace std;

int square(int& num) {
    return num * num;
}

int main(){
    int num = 5;
    int sq = square(num);
    cout << sq;
    return 0;
}
```

```
square(int&):
        mov     QWORD PTR [rsp-8], rdi
        mov     rax, QWORD PTR [rsp-8]
        mov     edx, DWORD PTR [rax]
        mov     rax, QWORD PTR [rsp-8]
        mov     eax, DWORD PTR [rax]
        imul    eax, edx
        ret
main:
        sub     rsp, 24
        mov     DWORD PTR [rsp+8], 5
        lea     rax, [rsp+8]
        mov     rdi, rax
        call    square(int&)
        mov     DWORD PTR [rsp+12], eax
        mov     eax, DWORD PTR [rsp+12]
        mov     esi, eax
        mov     edi, OFFSET FLAT:_ZSt4cout
        call    std::basic_ostream<char, std::char_traits<char> >::
        mov     eax, 0
        add     rsp, 24
        ret
```

Screenshoot 1: Pass by reference an int parameter

Passing a pointer:
Passing a pointer by value is similar to pass by reference: the value or the parameter registers are set to the memory addresses that hold the original value. Whereas in pass by a reference to a pointer, the parameter register holds memory address of a pointer that point to the actual parameter.



```cpp
#include <iostream>

using namespace std;

double inc(double* &d) {
    return ++*d;
}

int main(){
    double d= 9.6;
    double *d_ptr= &d;
    cout << inc(d_ptr);
    return 0;
}
```

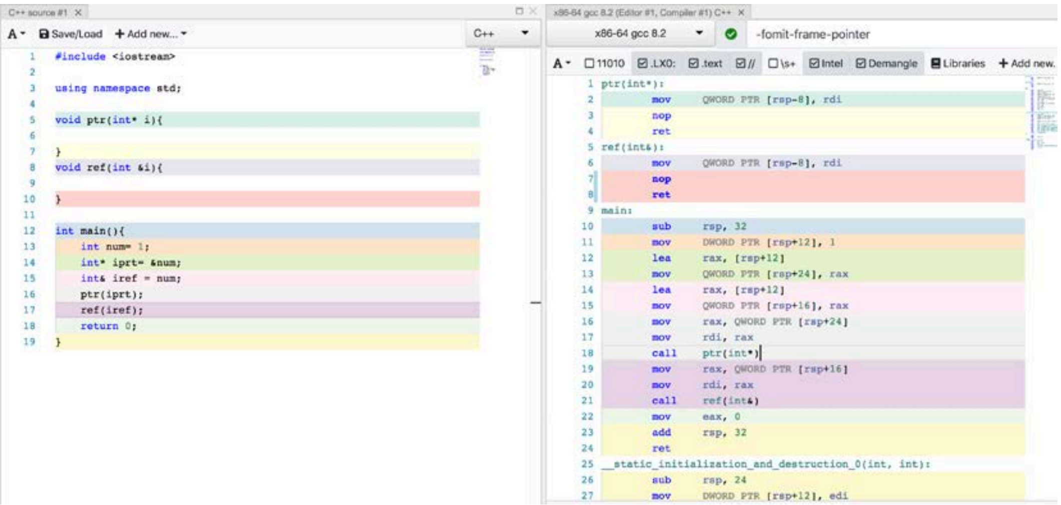
```
inc(double*&):
        mov     QWORD PTR [rsp-8], rdi
        mov     rax, QWORD PTR [rsp-8]
        mov     rax, QWORD PTR [rax]
        movsd   xmm1, QWORD PTR [rax]
        movsd   xmm0, QWORD PTR .LC0[rip]
        addsd   xmm0, xmm1
        movsd   QWORD PTR [rax], xmm0
        movsd   xmm0, QWORD PTR [rax]
        ret
main:
        sub     rsp, 24
        movsd   xmm0, QWORD PTR .LC1[rip]
        movsd   QWORD PTR [rsp+8], xmm0
        lea     rax, [rsp+8]
        mov     QWORD PTR [rsp], rax
        mov     rax, rsp
        mov     rdi, rax
        call    inc(double*&)
        mov     edi, OFFSET FLAT:_ZSt4cout
        call    std::basic_ostream<char, std::char_traits<char> >::
        mov     eax, 0
        add     rsp, 24
        ret
__static_initialization_and_destruction_0(int, int):
        sub     rsp, 24
        mov     DWORD PTR [rsp+12], edi
```

Screenshoot 2: Pass by reference a double pointer

2. Create a simple function that takes in an object. How are objects passed by value? How are they passed by reference? Specifically, what is contained in the parameter registers in each case?

When an object is passed by value, the compiler first creates a copy of that object inside the callee by calling the copy constructor. In order to call the copy constructor, the compiler needs to set values to the parameter registers beforehand. The values of these registers are copies of value of variables of the original object. After that, the caller sets the parameter registers to values of the copy of the original object that just made earlier by the copy constructor. Then the real callee is called. On the other hand, when an object is passed by reference, the caller sets the parameter register directly to the memory addresses that hold the values of the original object. This whole process implies the basic different of pass by value and pass by reference: the former creates a copy of the original and makes changes on that, while the latter changes the original directly.

Screenshoot 3: Object pass by reference



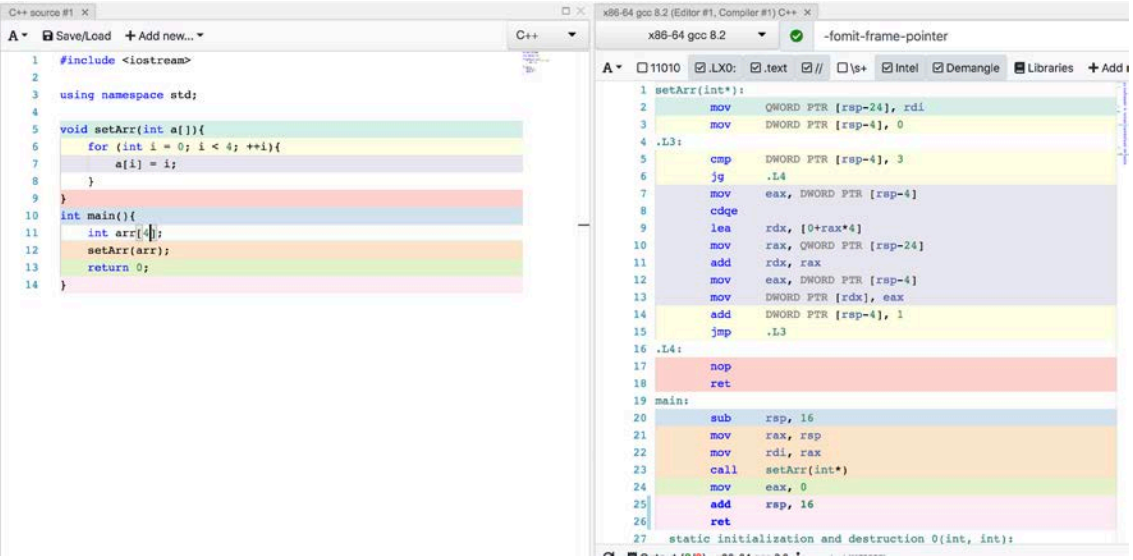
```
#include <iostream>

using namespace std;

void ptr(int* i){

}
void ref(int &i){

}

int main(){
    int num= 1;
    int* iprt= &num;
    int& iref = num;
    ptr(iprt);
    ref(iref);
    return 0;
}
```

```
ptr(int*):
        mov     QWORD PTR [rsp-8], rdi
        nop
        ret
ref(int&):
        mov     QWORD PTR [rsp-8], rdi
        nop
        ret
main:
        sub     rsp, 32
        mov     DWORD PTR [rsp+12], 1
        lea     rax, [rsp+12]
        mov     QWORD PTR [rsp+24], rax
        lea     rax, [rsp+12]
        mov     QWORD PTR [rsp+16], rax
        mov     rax, QWORD PTR [rsp+24]
        mov     rdi, rax
        call    ptr(int*)
        mov     rax, QWORD PTR [rsp+16]
        mov     rdi, rax
        call    ref(int&)
        mov     eax, 0
        add     rsp, 32
        ret
__static_initialization_and_destruction_0(int, int):
        sub     rsp, 24
        mov     DWORD PTR [rsp+12], edi
```

3. How are arrays passed into functions? How does the callee access the parameters? Where are the data values placed?

An array of type A is passed as an A pointer. When an array is passed into functions, the first parameter register rdi is set to the stack pointer that points to the memory segment that holds the array. The callee access the parameters by adding the rdi (now holds pointer to the memory address that holds the first element of the array). For example, the i-th element of an int can be accessed by rdi + i*a.

The data values is place on the stack with the first value in the smallest memory address and the last in the largest.



```
#include <iostream>

using namespace std;

void setArr(int a[]){
    for (int i = 0; i < 4; ++i){
        a[i] = i;
    }
}
int main(){
    int arr[4];
    setArr(arr);
    return 0;
}
```

```
setArr(int*):
        mov     QWORD PTR [rsp-24], rdi
        mov     DWORD PTR [rsp-4], 0
.L3:
        cmp     DWORD PTR [rsp-4], 3
        jg      .L4
        mov     eax, DWORD PTR [rsp-4]
        cdqe
        lea     rdx, [0+rax*4]
        mov     rax, QWORD PTR [rsp-24]
        add     rdx, rax
        mov     eax, DWORD PTR [rsp-4]
        mov     DWORD PTR [rdx], eax
        add     DWORD PTR [rsp-4], 1
        jmp     .L3
.L4:
        nop
        ret
main:
        sub     rsp, 16
        mov     rax, rsp
        mov     rdi, rax
        call    setArr(int*)
        mov     eax, 0
        add     rsp, 16
        ret
  static initialization and destruction 0(int, int):
```

Screenshoot 4: Passing array

4. Is passing values by reference different than passing by pointer? If they are the same, what exactly is passed in the parameter register? If they are different, how so?

They are the same. The memory address of the parameter is passed to the parameter register

Objects

1. How is object data laid out in memory? How does C++ keep different fields of an object “together"?

Object data is laid out next to each other on the stack with the first variable declared locates at the smallest (bottom)memory address.

2. Explain how data member access works for objects. How does the assembly know which data member to access?

To access certain data of an object, the compiler add the bytes used up by other data(s) of that object prior to the certain data to the base pointer.

3. How does method invocation work for objects? Specifically, how does the assembly know which object it is being called out of?

For each object being created, the compiler save a pointer to the memory segment of the object. So when a specific object is being called out of, the complier will pull out the pointer to that object’s data segment and access its data using the method discussed above.

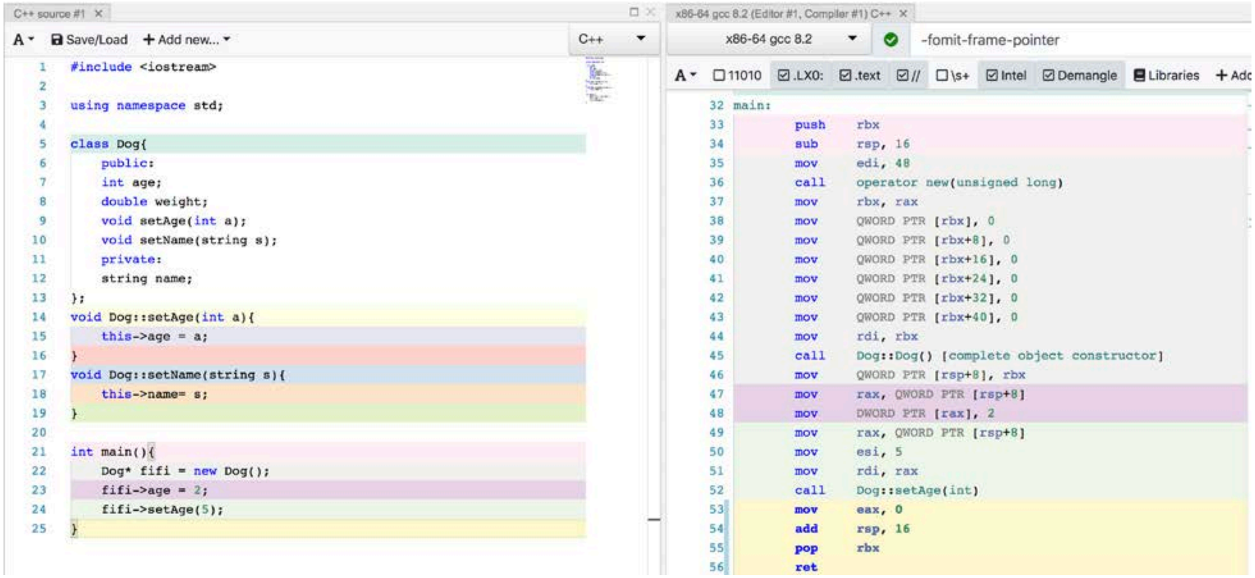
4. How are data members accessed both from inside a member function and from outside?

Regardless of inside or outside the member function, to access a data member, the assembly code needs the base address and information of how many data and of what type go before that data member. However, from inside a member function, there is no need to pass the object being called on, while from outside a member function, the assembly will not know which object is being referred to without an explicit mention.

5. How are public member functions accessed for your class? Call some of the public member functions for your class and examine the parameters. How is the "this" pointer

implemented? Where is it stored? When is it accessed? How is it passed to member functions?

If the name of a specific object is used to call the member function, the use of “this” is not necessarily. This is because when a member function is called using the object’s name, assembly can already figure which object is being referred to and pull out the right base address of that object. If used, “this” if the pointer to the data field of the object it is referring to. It is passed to a member function as the first parameter (rdi).



x86-64 gcc 8.2

-fomit-frame-pointer

```cpp
#include <iostream>

using namespace std;

class Dog{
    public:
    int age;
    double weight;
    void setAge(int a);
    void setName(string s);
    private:
    string name;
};
void Dog::setAge(int a){
    this->age = a;
}
void Dog::setName(string s){
    this->name= s;
}

int main(){
    Dog* fifi = new Dog();
    fifi->age = 2;
    fifi->setAge(5);
}
```

```asm
main:
        push    rbx
        sub     rsp, 16
        mov     edi, 48
        call    operator new(unsigned long)
        mov     rbx, rax
        mov     QWORD PTR [rbx], 0
        mov     QWORD PTR [rbx+8], 0
        mov     QWORD PTR [rbx+16], 0
        mov     QWORD PTR [rbx+24], 0
        mov     QWORD PTR [rbx+32], 0
        mov     QWORD PTR [rbx+40], 0
        mov     rdi, rbx
        call    Dog::Dog() [complete object constructor]
        mov     QWORD PTR [rsp+8], rbx
        mov     rax, QWORD PTR [rsp+8]
        mov     DWORD PTR [rax], 2
        mov     rax, QWORD PTR [rsp+8]
        mov     esi, 5
        mov     rdi, rax
        call    Dog::setAge(int)
        mov     eax, 0
        add     rsp, 16
        pop     rbx
        ret
```

Screenshoot 5: Calling member functions

Resources:

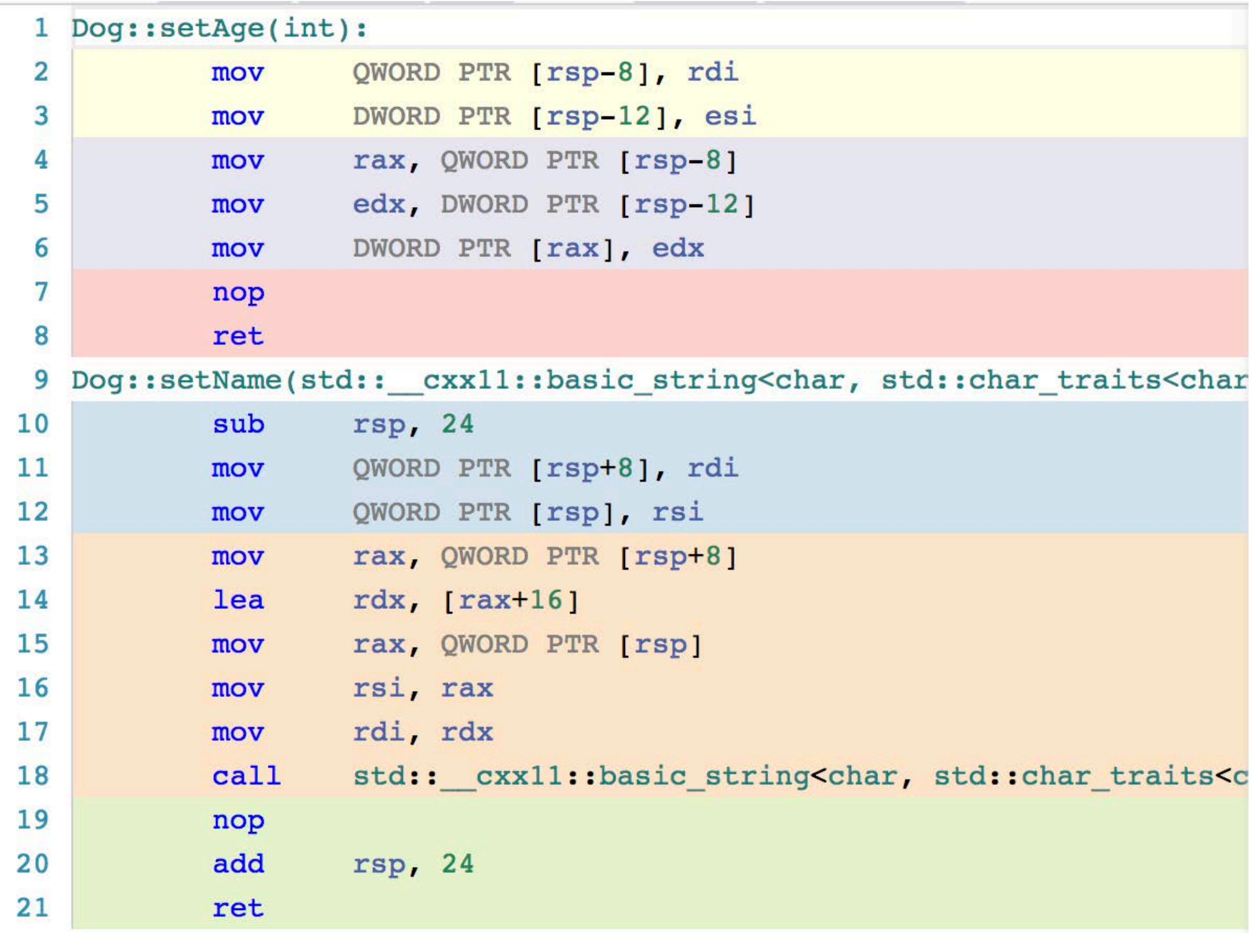
https://stackoverflow.com/questions/33556511/how-do-objects-work-in-x86-at-the-assembly-level

https://en.wikibooks.org/wiki/X86_Assembly/Print_Version#Memory

https://www.quora.com/How-many-registers-does-a-x86-64-processor-have

https://en.cppreference.com/w/c/language/struct

(I had many screenshoots to back up my answers but deleted most of them because of the size limit)



```
Dog::setAge(int):
        mov     QWORD PTR [rsp-8], rdi
        mov     DWORD PTR [rsp-12], esi
        mov     rax, QWORD PTR [rsp-8]
        mov     edx, DWORD PTR [rsp-12]
        mov     DWORD PTR [rax], edx
        nop
        ret
Dog::setName(std::__cxx11::basic_string<char, std::char_traits<char
        sub     rsp, 24
        mov     QWORD PTR [rsp+8], rdi
        mov     QWORD PTR [rsp], rsi
        mov     rax, QWORD PTR [rsp+8]
        lea     rdx, [rax+16]
        mov     rax, QWORD PTR [rsp]
        mov     rsi, rax
        mov     rdi, rdx
        call    std::__cxx11::basic_string<char, std::char_traits<c
        nop
        add     rsp, 24
        ret
```