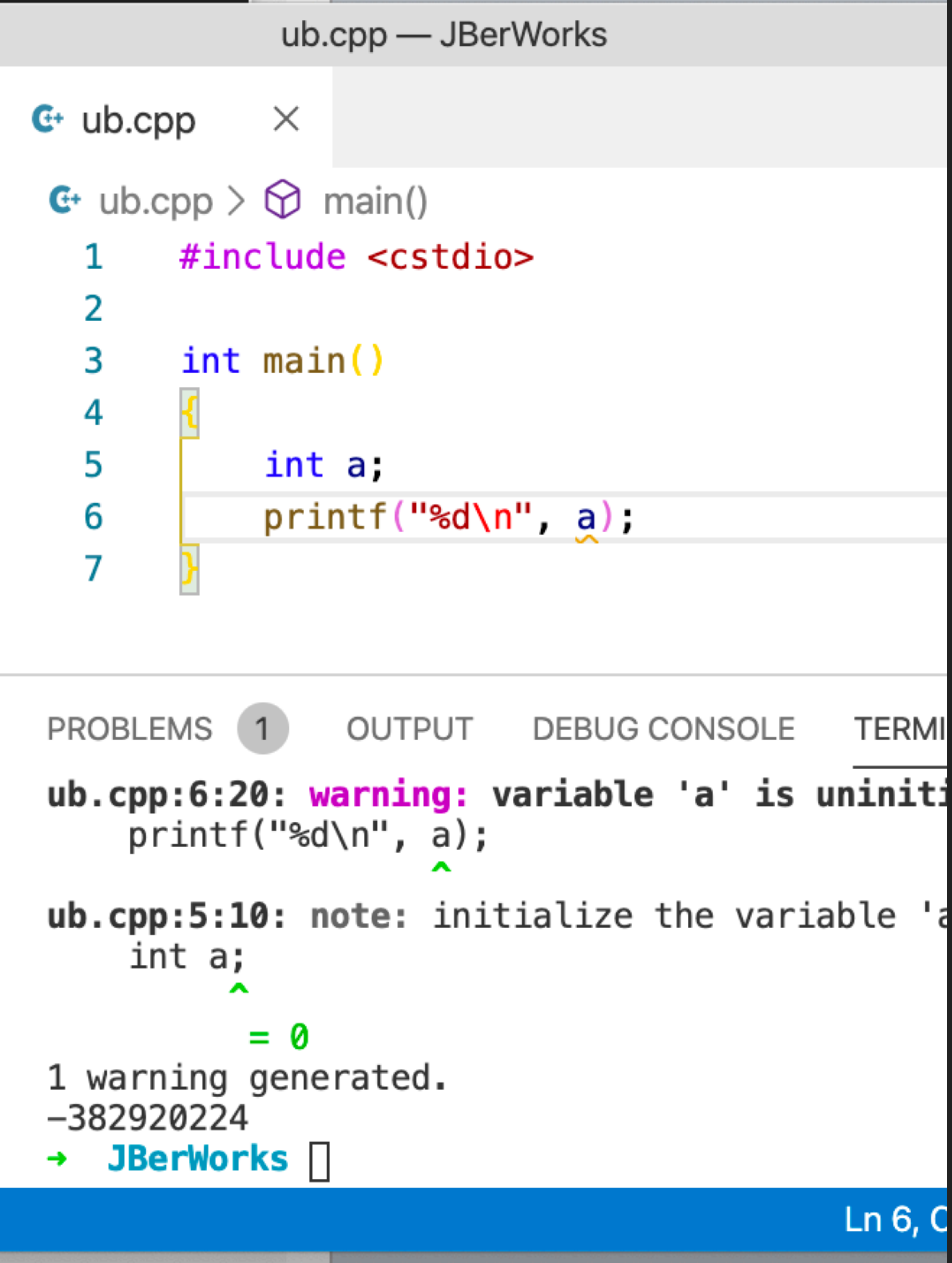
HOW TO WRITE

CPP CODE CORRECTLY



```cpp
#include <cstdio>

int main()
{
    int a;
    printf("%d\n", a);
}
```

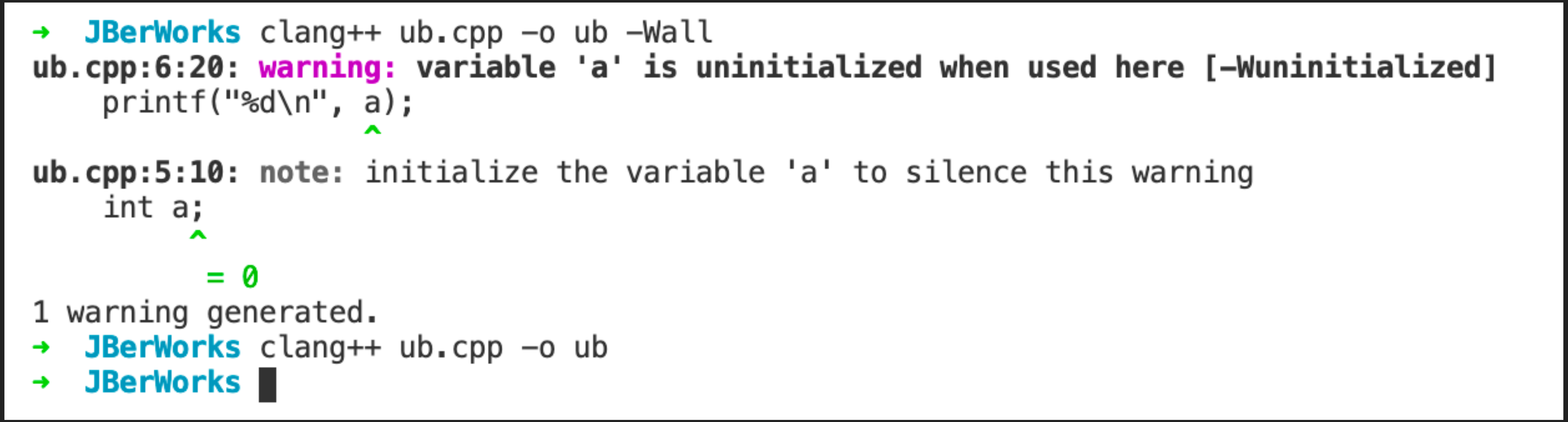
```
ub.cpp:6:20: warning: variable 'a' is uniniti
    printf("%d\n", a);
                   ^
ub.cpp:5:10: note: initialize the variable 'a
    int a;
         ^
          = 0
1 warning generated.
-382920224
➜  JBerWorks 
```

NO UNDEFINED BEHAVIOR

Undefined behavior (UB) is the result of executing a program whose behavior is prescribed to be unpredictable

HOW TO PREVENT UNDEFINED BEHAVIOR

- Use flag -Wall while compiling to get the warning information.



```
➜  JBerWorks clang++ ub.cpp -o ub -Wall
ub.cpp:6:20: warning: variable 'a' is uninitialized when used here [-Wuninitialized]
    printf("%d\n", a);
                   ^
ub.cpp:5:10: note: initialize the variable 'a' to silence this warning
    int a;
         ^
          = 0
1 warning generated.
➜  JBerWorks clang++ ub.cpp -o ub
➜  JBerWorks ▋
```

- Avoid warnings unless you are sure what you are doing
- You can even use flag -Wall -Werror, which to tell the compiler to treat all warnings as errors

GUARANTEE THAT STORAGE FOR STRINGS HAS SUFFICIENT SPACE

```
#include <iostream>
int main()
{
    char buf[12];
    std::cin >> buf;
}
```

```
#include <iostream>
#include <string>
int main()
{
    string buf;
    std::cin >> buf;
}
```

The easiest way: use string instead of char array

MEMORY MANAGEMENT (MEM)

- In OJ, try to use global variables and do not create new objects in the main function.
- If you need create objects in the main function, remember to delete it after use.
- Do not access freed memory.

PROPERLY DEALLOCATE DYNAMICALLY ALLOCATED RESOURCES

Allocator	Deallocator
`global operator new()/new`	`global operator delete()/delete`
`global operator new[]()/new[]`	`global operator delete[]()/delete[]`
`class-specific operator new()/new`	`class-specific operator delete()/delete`
`class-specific operator new[]()/new[]`	`delete[]()/delete[]`
`placement operator new()`	`N/A`
`allocator<T>::allocate()`	`allocator<T>::deallocate()`
`std::malloc(), std::calloc(), std::realloc()`	`std::free()`
`std::get_temporary_buffer()`	`std::return_temporary_buffer()`

DO NOT USE STD::RAND() FOR GENERATING PSEUDORANDOM NUMBERS

```
void f()
{
    std::string id("ID"); // Holds the ID, starting with the
                          // characters "ID" followed by a
                          // random integer in the range [0-10000].
    id += std::to_string(std::rand() % 10000);
    // ...
}
```

```
#include <random>
#include <string>
void f()
{
    std::string id("ID"); // Holds the ID, starting with the
                          // characters "ID" followed by a random
                          // integer in the range [0-10000].
    std::uniform_int_distribution<int> distribution(0, 10000);
    std::random_device rd;
    std::mt19937 engine(rd());
    id += std::to_string(distribution(engine));
    // ...
}
```

- The blue code uses the Mersenne Twister algorithm as the engine for generating random values and a uniform distribution to negate the modulo bias from the red code example.
- Modulo bias will cause some numbers in the range to never be available