John Gray

Mr. Perez

Programming Algorithms

12 December 2016

The Curse of Uninitialized Variables

Allowing programmers to use uninitialized variables is a big mistake on the part of language development. Such a mistake is easy to make and hard to trace, especially when running the program on different platforms. And the need for this feature is not present – the variable must always have a certain value.

Problem

Local variables, field variables, and others of a similar type are uninitialized variables. They will contain exactly what was written in the memory allocated for them during the initialization. In C + +, there are different rules for working with uninitialized variables, initialized variables, and null-initialized variables. It's a really complicated mechanism.

Considering the frequency of appearance of uninitialized variables in the code, you might think that tracking such situations is easy. This is not the case at all. Yes, of course, most variables are automatically initialized to zero, unless something else is indicated. However, this does not always happen. For example:

```
#include <iostream>


int main() {
```

```
    bool x;
    if( x ) {
        std::cout << "True" << std::endl;
    } else {
        std::cout << "False" << std::endl;
    }

    int y;
    std::cout << y << std::endl;
}
```

As a result of this code, you will always have False and 0 Errors related to a memory leak. But you can avoid using valgrind.

The main problem lies in the fact that the program continues to work even in this case. It reduces the probability of detecting the errors. Usually, it is possible to trace by only running the test on a different platform.

Why zero?

Why does the program automatically initialize variables as zero? In fact, it is performed not by the program, but by the operating system that simply does not allow the application to enter into an uninitialized memory area. It is the essence of the protective mechanism. If program A has fulfilled itself, and its results are somewhere in memory, program B does not detect them during their work, so the core self-cleans the memory.

Solution

A programming language should not allow the use of uninitialized variables. For example, the default value may be zero, regardless of how and in what scope we have created this variable.

At the same time, certain exceptions can be made due to optimization. For example, this occurs when working with low levels of memory. However, the optimizer usually finds unused variables, and cannot force them to initialize. If we need a block of uninitialized memory, we should be able to allocate its by ourselves. In this case, the programmer is aware of his activities, and therefore should not fall into the trap.

And to completely track all aspects with the memory initialization, the language must also have the special meaning NOINIT, which will show that the variable does not need to be initialized.

These changes should be implemented in the near standard of C++. The transformation of the previously uninitialized variables in the initialized will not affect the correctness of the execution of any program. This innovation will be fully backwards compatible, and will seriously improve the popular language.