SECURE AND EFFICIENT MANAGEMENT OF DEVICE MEMORY USING "DSRC_PTR" POINTER FOR INCREASE OF THE COMPETITIVENESS OF IT PRODUCTS IN THE MARKET

Armen SAFARYAN
Ph.D., Doctor of Economic Sciences
Martin MIRZOYAN
UESTC, CSE School, master's student
Vahagn GISHYAN
NPUA, ITTE Institute, SAED, master's student
Narek NALTAKYAN
NPUA, ITTE Institute, ISSD Chair, master's student
Aghasi SEYRANYAN
NPUA, ITTE Institute, ISSD Chair, master's student

Key words: smart pointers, system programming, dynamic memory, C++, efficient management

Introduction. In the fall of 2022 the US National Security Service published a report [National Security Agency, 2022, 1-7], the purpose of which was to warn IT companies to refrain from software tools implementation in C/C++ programming languages. According to the report, the main reason is the unsafe memory management in these languages [Detlefs, et al., 1994, 2-15]. The latter is a subtype of the resource management provided by the computer. Its main requirement is to allocate and deallocate dynamic memory on demand. This is strictly necessary for an arbitrary system where more than one process may be running at any given time. But this begs the following questions: What is the problem with manual memory management? What does that technical problem have to do with product competitiveness? Finally, if the problem is really so important that the organization has issued a separate report, why still those languages have not been replaced by safer alternatives? The main problem with manual memory management is that the responsibility for secure management rests on the programmer. Manual memory management provides a lot of flexibility, but does not guarantee a secure implementation. The consequences of mismanagement are unpredictable. They depend only from the virtual machine on which the software is running. Usually, a virtual machine provides minimal security mechanisms, without which the real machine can be harmed. However, these security mechanisms are only barriers to prevent software errors and external damage. Also, access to the entire range of memory addresses in languages can create technical vulnerabilities for hacker attacks [Nathan, et al., 2017.

It is clear that an IT product becomes uncompetitive in the market, if it has security problems. Nevertheless, there is another factor that does not allow these languages to be completely replaced with their safer alternatives. It is about efficient management. The main disadvantage of automated memory management is the additional costs that

are not present in manual management. IT products that need high performance for competitiveness often have to use C/C++. Thus, a dilemma arises: to process the products in an efficient or safe way. The following are the main problems of incorrect or inefficient memory management that occur during the software run:

1. Software tools (implemented, for example, in the C++ programming language) do not always have built-in tools that will inform about the problems that have arisen.
2. Existing third-party debugging tools (such as Valgrind [1]) are not guaranteed to be applied and are not always able to detect the problem at hand.

Methodology. The theoretical basis of the work consists the statements in response to the above report [Stroustrup, 2022, 1-2], speeches made at international conferences, researches of universities with world scientific qualifications, experiments of companies listed in international indexes, scientific works, reactions of standardization committees, etc. Safety and efficiency criteria are influential enough to substitute one for the other, and it is necessary to find a consensus between the two. For this purpose, a new tool was proposed in this article, which aims to make software implementation safer without losing its efficiency.

Literature review. There are a number of memory management methods, which can be conventionally divided into three groups: "manual", "fully automated" and "partially automated". An example of manual memory management is the C programming language, where the request, use, and release of memory are the responsibility of the programmer [Zlatanov, 2015, 3-6]. The approach has no technical limitations, which ensures high speed without spending time on side checks. However, this method does not have any security guarantee. In the framework of the article, we will call the pointers intended for manual memory management C-like (instead of “raw” pointer). The main part of fully automated memory management is commonly called garbage collection. The programmer requests and uses memory, but bears no duty and responsibility for freeing the memory. The latter is the problem of the virtual machine on which the software is running. A virtual machine collects information during memory allocation. At some point, the garbage collection starts in the virtual machine. Using the collected information, the garbage collector frees memory areas that are no longer in use. Thus, garbage collection ensures safe operation with memory. However, it is an expensive and long process, during which the software may temporarily stop working. The situation is different with the method of partially automated memory management. The abstractions that provide partial memory management are diverse. As a rule, such abstractions are called “smart” pointers and the latter provide certain security mechanisms due to certain limitations [Bukkapatnam, 2020, 2-9]. They can be implemented at the level of the language itself, or supported by standard or external libraries. Examples of such abstractions are the smart pointers *std::unique_ptr* and *std::shared_ptr* provided by the standard

library of the C++ programming language. Summarizing, we can say that in cases where high speed is needed, only manual or partially automated memory management is applicable. Fully automated memory management provides certain safety mechanisms through software compilation and run-time checks. Due to the additional checks during software runing, smart pointers run slower than similar pointers. For this reason, in a number of software tools where high efficiency is needed, C-like pointers are used, but they do not have security mechanisms.

Scientific novelty. The *dsrc_ptr* (debug smart, release c-like) pointer represents a partially automated memory management abstraction that behaves like a smart pointer in the debug phase, and behaves like a c-like pointer in the release phase.

dsrc ptr aims to solve the following problems:

1. Detect memory errors in an integrated way (without the use of extraneous means) during the debugging stage of the software implementation, prevent possible damages and inform the programmer about the existing errors.

2. Avoid the lack of efficiency due to absence of side checks in the final version of the software.

dsrc_ptr abstraction has C-like pointer behavior. A programmer's responsibilities include initialization of a pointer, freeing memory, providing checks, and organizing correct memory operations. The difference from the C-like pointer is that *dsrc_ptr* checks the correct use of the pointer during the debug phase, and is "replaced" by an identical C-like pointer in the release stage. The programmer can have full confidence in the safety and performance of working with the organized memory of the software implementation. The modernity of the pointer derives from the fact that the behavior of smart pointers in the C++ Standard Library does not change whether the implementation is in debug or release phase. From a technical point of view, the debug phase involves design, implementation and testing. In the release stage, it is necessary for the software to efficiently fulfill its main task. There is a fundamental difference between these two stages. *dsrc_ptr* takes into account the differences between the phases of the software implementations. The advantage of the pointer over currently available alternatives is that the main time wastage of *dsrc_ptr* is done during the debug phase of the software implementation. In the release stage, the pointer has no side costs and thus provides high efficiency. The following is the (implied) implementation of *dsrc_ptr* given in the C++ programming language [2]. The abstraction has some conventionalities because of technical limitations of the language:

1. *new/delete* operators have been replaced by *make/free* functions.

2. When declaring a pointer, the pointer must be allocated with memory, or defaulted to *nullptr*.

3. Immediately after freeing the memory, the pointer should be set to *nullptr*.

```
template<typename valueT>
class dsrc_ptr
{
private:
  using uint = unsigned int;
  /* impl */
private:
  valueT*           m_data;
#ifdef DEBUG
  uint*    m_ref_count = nullptr;
  bool*    m_is_deleted = nullptr;
#endif
};
```

A programmer's responsibilities include pointer initialization, freeing memory, ensuring of no double-allocation, etc. The pointer monitors these attempts and reports the problem. Error handling is also part of the programmer's responsibilities.

The change in pointer behavior is reflected in the implementation of abstraction. In addition to the main work, the methods have additional code blocks designed to provide checks during the configuration phase. For example, the postfix operator performs some checks in addition to its main purpose:

```
valueT& operator*()
{
#ifdef DEBUG
  if (m_data == nullptr || (m_is_deleted != nullptr && *m_is_deleted == true))
  {
          throw std::string("dsrc_ptr::operator* throw exception, trying indirect nullptr");
  }
#endif
  return (*m_data);
}
```

Examples of applications of abstractions are available on the mentioned website. The following are some presented cases. Case N1. Double-allocaion of the memory. Given the following function:

```
int main()
{
  int* ptr = nullptr;
  ptr = new int(0);
```

```
  ptr = new int(1);
  delete ptr;
}
```

The program will compile and run, but it has a memory leak. To detect the problem, you need to replace the pointers with *dsrc_ptr*.

```
int main()
{
  dsrc_ptr<int> ptr = nullptr;
  ptr = dsrc_ptr<int>::make(0);
  ptr = dsrc_ptr<int>::make(1);
  dsrc_ptr<int>::free(ptr);
}
```

The replaced version will also compile, but the program will break during execution because the pointer on line 3 is double-allocation.

```
int main()
{
  dsrc_ptr<int> ptr = nullptr;
  ptr = dsrc_ptr<int>::make(0);
  dsrc_ptr<int>::free(ptr);
ptr = dsrc_ptr<int>::make(1);
  dsrc_ptr<int>::free(ptr);
}
```

After the memory error is fixed, the program can be compiled as a release version and will not have any memory loss on side errors. Case N2. The problem of *shared_ptr*:

dsrc_ptr and *shared_ptr* are very similar. Both enable partially-automated dynamic memory management. Both allow multiple pointers to access a single address. However, *shared_ptr*, unlike *dsrc_ptr*, does automatic indirection of the memory.

```
class SomeT
{
    std::shared_ptr<SomeT> m_ptr;

public:
    SomeT() : m_ptr(nullptr) {}

static bool tying(std::shared_ptr<SomeT>& h1, std::shared_ptr<SomeT>& h2)
    {
        if (h1 == nullptr || h2 == nullptr)
            return false;
```

```
            h1->m_ptr = h2;
            h2->m_ptr = h1;

            return true;
        }
};

int main()
{
  std::shared_ptr<SomeT> ptr1 = std::make_shared<SomeT>();
  std::shared_ptr<SomeT> ptr2 = std::make_shared<SomeT>();
SomeT::tying(ptr1, ptr2);
  //memleak
            return 0;
}
```

A memory leak will occur. The problem is that the memory is freed when there are no more memory observers. In the example, a recursive connection is implemented and the memory cannot be freed. The example is famous and when using *shared_ptr*, it is recommended to use *weak* pointer to avoid such possible problems.

dsrc_ptr does not have such a problem. Replacing *shared* pointer in the above example with *dsrc_ptr* will solve the memory leak problem.

```
class SomeT
{
    dsrc_ptr<SomeT> m_ptr;

public:
    SomeT() : m_ptr(nullptr) {}

    static bool tying(dsrc_ptr<SomeT>& h1, dsrc_ptr<SomeT>& h2)
    {
        if (h1 == nullptr || h2 == nullptr)
            return false;

        h1->m_ptr = h2;
        h2->m_ptr = h1;

        return true;
    }
```

```
};

int main()
{
    dsrc_ptr<SomeT> ptr1 = dsrc_ptr<SomeT>::make();
    dsrc_ptr<SomeT> ptr2 = dsrc_ptr<SomeT>::make();

    SomeT::tying(ptr1, ptr2);

    dsrc_ptr<SomeT>::free(ptr1);
    dsrc_ptr<SomeT>::free(ptr2);

    return 0;
}
```

Analysis. The IT sector is considered strategic in a number of countries. The reason is that the development of IT products does not require raw materials, special equipment or a large amount of labor. The conditions are not harsh, they are not destructive from an environmental point of view, and, on the contrary, they can bring great profits and develop the economy. Such positive conditions lead to the creation of new IT companies, as a result of which there is a competition between them in the market, where those companies win whose products provide the greatest comfort and confidence for the user. It is not possible to be successful in the market if the product has performance issues or the user refuses to use it due to some security issues. Proper memory management in C++ can have a significant impact on the competitiveness of products, especially in software applications where performance and reliability are important factors. Here are a few ways in which proper memory management in C++ can affect the competitiveness of products:

1. Improved Performance: Proper memory management can help avoid memory leaks and fragmentation, which can cause slowdowns and crashes in the program. By optimizing memory usage and reducing unnecessary memory allocation, a program can run faster and more efficiently. This can be especially important in high-performance applications, such as video games or scientific simulations.

2. Enhanced Security: Poor memory management can lead to buffer overflows and other security vulnerabilities. By implementing proper memory management techniques, such as using smart pointers and avoiding pointer arithmetic, a program can be made more secure and less vulnerable to attacks.

3. Reduced Costs: Memory leaks and other memory-related bugs can be difficult to diagnose and fix, leading to increased development and maintenance costs. Proper

memory management can help avoid these issues and reduce the overall cost of development.

4. Increased Reliability: Memory-related bugs can also cause crashes and other errors, leading to frustrated users and lost productivity. Proper memory management can help prevent these issues and make a program more reliable and stable, which can improve customer satisfaction and lead to greater sales.

In summary, proper memory management in C++ can have a significant impact on the competitiveness of products, improving performance, security, reliability, and reducing costs. However, according to the theory of improvements, an arbitrary improvement is made at the expense of certain losses. In particular, the checks are performed only during the debugging, and they are absent in the release stage. Therefore, if a problem occurs at that stage, the *dsrc_ptr* pointer will not be able to debug and report it. In addition, the security of the software using such a tool directly depends on the tests. If the software is not tested, then it is meaningless to talk about the efficiency of the tool.

High quality is a crucial factor in market competitiveness. When a product or service is of high quality, it meets or exceeds customer expectations and needs. This can lead to increased customer satisfaction and loyalty, as well as positive word-of-mouth marketing. When customers are satisfied with a product or service, they are more likely to become repeat customers and recommend it to others. This can lead to increased sales and market share.

Thus, high quality can be a key differentiator in the market, allowing a company to stand out from competitors and attract more customers. By prioritizing quality, a company can increase customer satisfaction, build a positive reputation, command higher prices, and reduce costs, all of which can contribute to increased market competitiveness.

Safety and performance are two critical factors that can significantly impact the quality of a product. Safety is essential for products that can cause harm, while performance can affect customer satisfaction and product reliability. Ensuring safety and performance can lead to high-quality products that meet customer needs and expectations.

Conclusion. A partially-automated memory management abstraction called *dsrc_ptr* has been developed. It is designed for all software implementations that require high performance. The pointer is a convenient and easy tool to find out the existence of memory errors in the software during tests without additional side checks. The main drawback of the pointer is that it detects the problem only in the stage of the software run. This means that the pointer is directly dependent on tests. If the tests during the debugging phase of the software implementation did not fully cover the possible cases, then the device may contain all possible memory errors. In devices requiring high-speed performance, *dsrc_ptr* can be used instead of C-like pointers.

References:
1. https://valgrind.org/
2. https://github.com/VahagnGishyan/ScientificArticles/tree/master/dsrc_ptr/impl
3. National Security Agency | Cybersecurity Information Sheet, U/OO/219936-22 | PP-22-1723 | NOV 2022 Ver. 1.0
4. Bjarne Stroustrup, “A call to action: Think seriously about “safety”; then do something sensible about it”, Columbia University, Doc. no. P2739R0 Date: 2022-12-6
5. https://cppcon.org/
6. Detlefs, D.; Dosser, A.; Zorn, B. "Memory allocation costs in large C and C++ programs" (PDF). Software: Practice and Experience. 24 (6): 527–542. CiteSeerX 10.1.1.30.3073. doi:10.1002/spe.4380240602. S2CID 14214110
7. https://thephd.dev/your-c-compiler-and-standard-library-will-not-help-you
8. Krishnaveni Bukkapatnam, Smart Memory Management (SaMM) For Embedded Systems without MMU, et al 2020 IOP Conf. Ser.: Mater. Sci. Eng. 981 032010
9. Zlatanov Nikola, 2015, Dynamic Memory Allocation and Fragmentation. ESC 182
10. Nathan Burow, Derrick McKee, Scott A. Carr, Mathias Payer, Comprehensive User-Space Protection for C/C++, 17 Apr 2017, arXiv:1704.05004

Armen SAFARYAN, Martin MIRZOYAN, Vahagn GISHYAN,
Narek NALTAKYAN, Aghasi SEYRANYAN

Secure and efficient management of device memory using “dsrc_ptr” pointer for increase of the competitiveness of IT products in the market

Key words: smart pointers, system programming, dynamic memory, C++, efficient management

A smart pointer for partially automated control of dynamic memory called *dsrc_ptr* has been developed. It is designed to detect memory errors, as a result of which IT products will become more competitive in the market. One of the clearest indicators of the relevance and importance of the problem is the report released by the US National Security Agency in the fall of 2022, which warned against developing software that may have similar problems. The presented pointer took into account the mentioned problems. It can be implemented both at the level of the programming language and provided by the library. The pointer provides a toolkit for detecting memory errors in software during debugging. No third-party tools will be needed to locate and fix memory errors. In the release version of the software implementation, the pointer gets a similar pointer behavior and has no speed loss beyond the programmer's settings. The advantage of the proposed changes over the existing alternatives is that the pointer independently detects errors during the debugging process, and does not incur additional costs in the release version. A software implementation of the proposed concept is given in C++ programming language.