Technical Report on C++ Performance

In this lesson, we will discuss the C++ features, their overhead, and usage in light of Technical Report on C++ Performance.

WE'LL COVER THE FOLLOWING

- C++ Features, Overhead, and Usage

The Working Group WG 21 published the ISO/IEC TR 18015, a document that is the ultimate source on the performance numbers of the C++ features. The document expresses its concerns directly.

- To give the reader a model of time and space overheads implied by use of various C++ language and library features,
- To debunk widespread myths about performance problems,
- To present techniques for use of C++ in performance applications, and
- To present techniques for implementing C++ Standard language and library facilities to yield efficient code.

The paper documents the work of experts like Dave Abrahams, Howard Hinnand, Dietmar Kühl, Dan Saks, Bill Seymour, Bjarne Stroustrup, and Detlef Vollmann.

C++ Features, Overhead, and Usage

The authors of the Technical Report on C++ Performance used three computer architectures with five different compilers for their analysis. They used compilers with different optimization options. Let's go over why the results of the analysis were quite remarkable.

- **Namespaces**
  - They have no significant overhead in size and performance.

- **Type converting operator**
  - The C++ casts `const_cast`, `static_cast`, and `reinterpret_cast` differ neither in size nor in performance from their C predecessor.
  - However, `dynamic_cast`, which is executed at run time, has some overhead. (Remark: The conversion has no C predecessor.).
- **Inheritance**
  - Class
    - A `class` without virtual functions is as big as a `struct`.
    - A class with virtual functions has the overhead of a pointer and a virtual function table. These are about 2 to 4 bytes.
  - Function calls
    - The call of a non-virtual, non-static, and non-inline function is as expensive as the call of a free function.
    - The call of a virtual function is as expensive as the call of a free function with the help of a pointer that is stored in a table.
    - Virtual functions of a class template can cause overhead in size. (Remark: Functions that do not depend on template parameters should be extracted in a base class. Therefore, the functionality - independent of template parameters - can be shared between all derived class templates.)
    - The *inlining* of a function causes significant performance benefits and is close to the performance of a C macro.
  - Multiple Inheritance
    - Can cause time and/or space overhead. Virtual base classes have overhead compared to non-virtual base classes.
- **Run-time type information (RTTI)**
  - There are about 40 additional bytes required for each class necessarily.
  - The `typeid` call is quite slow due to the quality of the implementation.
  - The conversion during runtime with `dynamic_cast` is slow, likely due to the quality of the implementation.

- **Exception handling**
    - There are two strategies for dealing with exceptions: the code and the table strategy. The coding strategy has to move and manage additional data structures for dealing with exceptions. The table strategy has the execution context in a table.
        - The coding strategy has a size overhead for the stack and the runtime. The runtime overhead is about 6%. This overhead exists even without the throwing of an exception.
        - The table strategy has neither overhead in program size nor in runtime. (Remarks: That statements hold only if no exceptions were thrown.). The table strategy is more difficult to implement.
- **Templates**
    - For each template instantiation, you get a new class template or function template. Therefore, the naive use of temples can cause code bloat. Modern C++ compilers can massively reduce the number of template instantiations. The usage of partial or full specialization helps to reduce the template instantiation.

You can read other details of the report here TR18015.pdf.

It is important to note that the “Technical Report on C++ Performance” is from 2006. In the case of modern C++, there are many features for writing faster code that are not accounted for in the report. Author Detlef Vollmann states that they plan to update the report to modern C++, though there are some hurdles to overcome.

Now that we have a solid background in C++, let’s go over the specifics of this coding language. In the next chapter, we will talk about the **Safety-Critical Systems** in Embedded Programming.