"Vasile Alecsandri" University of Bacău
Faculty of Sciences
Scientific Studies and Research
Series Mathematics and Informatics
Vol. 34 (2024), No. 1, 57 - 62

NEW UTILITY LIBRARIES FOR LINUX WITH COMPLEX DATA TYPES AND SYSTEM RESOURCES

ALEXANDRU PINTEA

Abstract. Operating systems have different limitations for every primitive data-type regarding the maximum value that can be stored. For numeric data types, those limitations could be eliminated by converting to `string`. New C++ utility libraries are provided for both complex data types and system resources. One of the introduced C++ libraries is able to perform operations with numbers stored in string. Another, stores data in its self defined data type. The last one provides Linux compatibility for C++, taking care of various C++ function definitions.

1. Introduction and preliminaries

Data types have always been a part of programming and lately they have become increasingly customisable. The innovation that led to them becoming more customisable is of course Object Oriented Programming, OOP; even so, that does not solve all issues that storing data might pose, since various limitations of the primitive data-types are still in effect. With open-source operating systems, every application is welcome and those that have a terminal-based UI are most cross-platform.

Most primitive data types are stored on a memory space that consists of a limited number of bytes, which limits the maximum amount of data that can be stored [1].

Keywords and phrases: code optimization, programming.
(2020)Mathematics Subject Classification: 97R70, 68N19, 97R10

That seems to apply more to numeric data types rather than to `string` data types. Therefore, if more digits need to be added to a number that could be done by converting them to a string for later processing. That is precisely what the new `string` C++ library, `strings` does; it computes operations with numbers stored in strings [2].

To extend the capabilities of OOP data types, the introduced `multi` library provides the user with a data type that can store any number of variables in a `string`. While the improved `string` C++ library, `strings` converts numbers to strings and performs operations with them, the `multi` library provides one data type that can store multiple data types in variables that can be added and deleted (unlike with OOP declarations). Therefore, `multi` library stores data-types e.g. `int`, `bool`, `string` and `double` in a string [2].

As a pioneering programming language (which led to the OOP revolution), C++ does not aim to provide easy-to-use bindings with operating system commands, in order to let the user have more flexibility. However, when communicating with the operating system becomes a constant necessity, certain functions need be made to respond to such a need.

The new `lnux` C++ library provides the user with functions that use the Linux operating system to easily perform operations native to the system command line. Even so, conversions and validations are performed to make sure that the user is able to perform these actions smoothly, avoiding system errors [2].

In order to reason about low-level programming a semantic that "can deal with programs that are not statically type-correct" is required [1]. That shows that type does not always matter, meaning that using numbers stored in strings can be a sustainable and effective choice for large numbers. The C++ API, Data Type Encoding Language (DTEL) "fully leverages the C++ type system to catch many encoding errors at compile time" [3]. Such error signalising is incorporated in `multi` library, which is able to validate available data types and names [2].

C++ 11 was improved in [4] with an implementation of tensors of arbitrary rank. In order to provide convenient mathematical syntax an expression template was introduced based on the array class template. Therefore, C++ language gets another layer "when dealing with algebraic objects and their operations, without compromising performance" [4].

Performance as well as elimination of limits provided by primitive data types, is essential for such algebraic objects. Therefore using numbers stored in strings is preferable when dealing with extremely large numbers (or lots of decimals). In a recent C++ book [8] operator-overloading facilities are illustrated. There are other programming languages (e.g. Python [6]) which have already implemented arbitrary-length integers nativly.

In [5] was introduced a new library defining a function that converts a string representing a real number to an interval according to a specific algorithm. Such conversions were implemented here for `lnux` and `multi` libraries, to convert string data [2].

As Linux kernel grows in size it has maintenance problems. An object-oriented wrapper based approach to Linux kernel providing OO abstractions to external modules was proposed in [7]. The proposal is beneficial in terms of reusability and extensibility. The C++ `multi` library provides such an OO abstraction, by also letting the user customise the variables provided [2].

2. Main results

To facilitate the work with complex data types in C++ and the efficient use over the Linux shell, new libraries are presented: `multi`, `strings` and `lnux` [2].

`multi` library

The `multi` library can keep variables in a string, enabling it to keep as many variables as needed. It provides functions that can be used to verify if a variable is available in the string, as well as getting / setting it. Other provided functions include implementations for leaving only numeric, symbols or alphabetic characters in a string, as well as converting any of the available data types (`long`, `bool`, `double`) to and out of the string.

The `multi` class facilities include adding or getting multiple (`int`, `long`, `double`, `float`, `bool`, `char` or `string`) variables to/out from the `multi_string`. It can also save/load it's `multi_string`, which contains all the variables.

The `multi_string` can be stored in a file for further use, reduce the whitespace of a string and use special identifiers (words, numeric, and chars) for the string data-type to separate alphabetic, numeric and symbolic characters. It is able to store arrays too.

```
long long_a = 19999;
multi_1.add(long_a, get_name (long_a) );
multi_1.get("long_a", "long", long_a);
cout<<long_a;
multi_1.del("long_a", "long");
```

Convert the given string to multiple variables:

```
void make_types ()
{
string data=this->multi_string;
this->multi_string="";
this->multi_string=this->multi_string+"words string "
+ leave_words(data) +"\n";
this->multi_string=this->multi_string+"numerics double "
+ leave_numerics(data)+"\n";
this->multi_string=this->multi_string+"chars char "
+ leave_chars(data)+"\n";
}
```

Example of a `multi _string` (encoding: variable name or data-type or data - array or not)

```
a_int int 10 9 3 2 4
a_bool bool true
a_double double 1.07 9.1 -3.024 2.0099999 4.1
a_float float 1.7 9.1 3.0239999 2.0099999 4.0999999
a_char char a b l r
a_string string ab b l r
long_a long 19999
```

Example of error notification: e.g. a variable name is already set: `long_a long`.

`strings` library

The `strings` library has functions for adding or subtracting two numbers (`integer` or `double`). It can also check if the string represents an even / odd / positive or negative number.

While adding / subtracting char-stored digits, it can tackle reminders and preserve the size of the given numbers by not ever converting to any data-type besides string. In other words, even when separating the decimal and integer part of a `double`, the `string` data-type is used to both give and receive data. Such numbers can have any length, in the computation limits.

Most of its functions use multiple cases to decide how to perform the addition or subtraction or other operations (decide which number is

larger, remove decimals, remove sign, and decide if the number has decimals). For the addition and subtraction functions, conditions for different types of numbers are included.

`lnux` library

The proposed library called `lnux` is a C++ library for Linux that makes it easier for C++ to communicate with the command line. It is able to decide if a file or folder is at a location, give C++ the USER and PATH of the Linux system, determine filesize of files or folders, count lines of a file, and give C++ every filename within a certain folder. It uses the C++ `convert` library from [5]. The library also has C++ functions to perform basic commands as `mkdir, chown, chmod, rm, cp, ls` and even `xrandr` (used for screen brightness).

Verifications are done when needed; for example to validate the fact that a folder to be created with `mkdir` does not already exist and also decide if the permission that is given to `chmod` is valid. It also verifies for the availability of files or folders before attempting to determine their size (using `is_folder()` and `is_file()` functions that check every filename in a certain folder to find the required one). It communicates with the Linux system by outputting input data to a file, executing Linux commands and then getting the results out of a file that was initially used for input:

```
string command_size_folder="\nSIZE=$(du -s \"$FILE\")
\nprintf $SIZE >> \"$PWD/cpp-sh\"";
char command_linux[] = "IFS= read -r
filepath < \"files-ls\"\ncd $filepath\nls_data=$( ls -m )
\ncd ../\nrm -R files-ls\nfor value in $ls_data\ndo
\nprintf $value >> files-ls\ndone";
```

Commands are given either using a char array, or through conversion of the string data-type to a char array, since the C++ command only takes char arrays for an argument.

3. Conclusions

Libraries are useful for performing certain actions efficiently. Non-primitive data types provide a good format for modeling any application data (as in the `multi` and `strings` libraries). System compatibility libraries (such as `lnux`) enable programming languages to perform system-related operations seamlessly. The presented libraries enable their users to extend C++ capabilities beyond the limitations primitive data types, or beyond the scope of the C++ environment, towards system functionalities.

References

[1] M.Hohmuth and H.Tews, **The Semantics of C++ Data Types: Towards Verifying low-level System Components**, Conference: Theorem Proving in Higher Order Logics (TPHOLs) (2003), 127–144.

[2] A. Pintea. Code, online at Github, https://github.com/AlexandruPintea2000/Cpp-libraries-applications/Libraries/

[3] J.Graham, **Creating an HLA 1516 Data Encoding Library using C++ Template Metaprogramming Techniques**, Spring Simulation Interoperability Workshop Proceedings, (2007), 07S-SIW-035.

[4] A.M.Aragon, **A C++11 implementation of arbitrary-rank tensors for high-performance computing**. Comp.Phys.Comm. 185, 6(2014), 1681–1696.

[5] M.A. Jankowska, **C++ Library for Floating-Point Interval Arithmetic. User and Reference Guide.** Poznan University, Tech.Rep. 2018.

[6] Lindstrom, G.. (2005). **Programming with Python.** IT Professional. 7,5 (2005), 10–16.

[7] V. Reddy, **Object-oriented wrappers for the Linux kernel**. Software-Practice and Experience, 38(2008),1411–1427.

[8] A.Singht, P.Manisha, K.Renu. (2017). **C++ Programming - Principles of Design an Implementation.** Mirdashti Publisher, 2017.

Coventry University
Priory Street, Coventry, CV1, UK
e-mail alexandru.pintea@ieee.org