3-gsl-utilities

June 3, 2024

1 Miscellaneous GSL utilities

Some isolated utilities, which sometimes prefigure features to come in the next versions of the C ++ standard and its library.

1.1 "C-style" strings `gsl::zstring` and `gsl::czstring`

The two aliases below simply allow to draw attention to strings that are supposed to end with a null character: * `zstring`: `char *`, being `nullptr` or pointing to a C-style string; * `czstring`: `char const *`, being `nullptr` or pointing to a C-style string.

If your C-style string pointer should not be null, use `gsl::not_null<zstring>`.

If your string is not supposed to end with a null character, use `string_view`.

1.2 For generic and efficient access to strings: `std::string_view`

The `std::string` class is both very convenient and very expensive.

On the model of `gsl::span`, a *view* was proposed for string, renamed into `std::string_view` upon integrating into C++17.

Composed of a pointer and a size, the `std::string_view` class allows to create easily and quickly a **constant non-owner view to a sequence of contiguous characters in memory**: not just an `std::string`, but also a simple literal string or a `Qstring` (Qt).

As `string_view` stores the size of the string, the underlying string does not necessarily need to end with a character `0`.

Writing a function receiving a `string_view` (rather than a `std::string const &`) has benefits:

- more efficiency than `std::string` with a literal string;
- capacity of receiving all kinds of strings as input;
- efficient creation of a substrings because the final `0` character is no longer necessary.

Be careful: the `string_view` does not own or duplicate the underlying data. Be careful not to use a view when its underlying data has disappeared.

In addition, by going through a `string_view`, we lose the guarantee of having the final `0`. If the string must then be passed to other functions that expect the terminal `0`, it is better to use `string` from the beginning.

1.3 Contracts

GSL offers two dedicated assertions to validate *pre-conditions* and *post-conditions*. * `Expects(p)`: stop the application unless `p == true`. * `Ensures(p)`: stop the application unless `p == true`.

These assertions are **currently implemented via macros**, and must be placed in the body of functions, while waiting for future decisions of the standardization committee that deals with contracts and syntax of assertions.

In the Contract Proposal, it is proposed to to move these declarations to the level of function declarations and use some specifiers such as `[[expects: p]]`.

The contracts are not yet in C++20.

[6]:

```
%%file tmp.gsl-utilities.cpp

#include <iostream>
#include <gsl/gsl>

void divide( double num, double den ) {
    Expects(den!=0) ;
    std::cout<<(num/den)<<std::endl ;
}

int main() {
    divide(5,0) ;
}
```

```
Overwriting tmp.gsl-utilities.cpp
```

[7]:

```
!rm -f tmp.gsl-utilities.exe && g++ -std=c++17 -I./ tmp.gsl-utilities.cpp -o␣
 ↪tmp.gsl-utilities.exe
```

[8]:

```
!./tmp.gsl-utilities.exe
```

```
terminate called without an active exception
Aborted (core dumped)
```

1.4 Final action: `finally()`

As a last resort, when the resources management tools are not sufficient, we can define a function to be invoked at the end of a block.

[5]:

```
%%file tmp.gsl-utilities.cpp

#include <iostream>
#include <gsl/gsl>

struct Demo {
    Demo() { std::cout<<"Constructor"<<std::endl ; }
    ~Demo() { std::cout<<"Destructor"<<std::endl ; }
} ;

int main() {
    Demo * d {new Demo} ;
    auto _ { gsl::finally( [d](){ delete d ; } ) } ;
    // ...
}
```

```
Overwriting tmp.gsl-utilities.cpp
```

[3]:

```
!rm -f tmp.gsl-utilities.exe && g++ -std=c++17 -I./ tmp.gsl-utilities.cpp -o tmp.gsl-utilities.exe
```

[4]:

```
!./tmp.gsl-utilities.exe
```

```
Constructor
Destructor
```

1.5 Numerical utilities: `narrow_cast<T>(x)` and `narrow<T>(x)`

The first one, `narrow_cast<T>(x)`, is just a statement for programmers or testing tools. It clarifies the fact that the developer wants **voluntarily** to force a value to a less precise type.

The second one, `narrow<T>(x)`, also checks at runtime that the value `x` was not modified when its type was transformed into a `T`, and throws an exception (or terminates the program) otherwise (if `static_cast<T>(x) != x`).

[1]:

```
%%file tmp.gsl-utilities.cpp

#include <iostream>
#include <gsl/gsl>

int main() {
    double d { 3.14 } ;
    int i { gsl::narrow<int>(d) } ;
    std::cout<<i<<std::endl ;
}
```

```
Writing tmp.gsl-utilities.cpp
```

[2]:

```
!rm -f tmp.gsl-utilities.exe && g++ -std=c++17 -I./ tmp.gsl-utilities.cpp -o tmp.gsl-utilities.exe
```

[3]:

```
!./tmp.gsl-utilities.exe
```

```
terminate called without an active exception
Aborted (core dumped)
```

2 Questions ?

2.1 Exercise

Create your own minimal `my_narrow`, so that the program below accept the narrowing of `42`, but will crash when trying to narrow `3.14`.

[11]:
```
%%file tmp.gsl-utilities.cpp

#include <iostream>
#include <cassert>

// ... PUT HERE YOUR IMPLEMENTATION OF my_narrow  ...

int main()
 {
  double d1 {42} ;
  int i1 {my_narrow<int>(d1)} ;
  std::cout<<i1<<std::endl ;

  double d2 {3.14} ;
  int i2 {my_narrow<int>(d2)} ;
  std::cout<<i2<<std::endl ;
 }
```

```
Overwriting tmp.gsl-utilities.cpp
```

[ ]:
```
!rm -f tmp.gsl-utilities.exe && g++ -std=c++17 -I./ tmp.gsl-utilities.cpp -o␣
 ↪tmp.gsl-utilities.exe
```

[ ]:
```
!./tmp.gsl-utilities.exe
```

2.2 Sources

- https://stackoverflow.com/questions/40127965/how-exactly-is-stdstring-view-faster-than-const-stdstring
- http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#gsl-guidelines-support-library
- http://modernescpp.com/index.php/c-core-guideline-the-guidelines-support-library
- http://nullptr.nl/2018/08/refurbish-legacy-code/