std::string::resize_and_overwrite

Tobias Hoffmann

C++ User Gruppe Aachen, 2024-02-07

Motivation

- Daniel Frey (2023-11-29): Accessing Private Members the Right Way

Motivation

- Daniel Frey (2023-11-29):
  Accessing Private Members the Right Way

```
// this pattern occurs in https://github.com/taocpp/taopq
std::string read( const std::size_t max )
{
  std::string buffer;
  buffer.resize( max ); // slow
  const std::size_t delivered = read_from_database( buffer.data(), max );
  buffer.resize( delivered );
  return buffer;
}

// often ‘max’ is large (e.g. read up to 1MB), while ‘delivered’ is often smal
// leading to the first ‘resize’ to be slow, as the string will always be
// initialised, i.e. filled with ‘\0’.
```

MOTIVATION

Proposed solution:

```
template< typename T >
void resize_uninitialized( std::basic_string<T>& v, const std::size_t n )
{
  if( n <= v.size() )
  v.resize( n );
  else {
    if( n > v.capacity() )
      v.reserve( n );
    resize_uninitialized_impl( v, n );
    // basically: calls private
    //   v._Eos(n) [MSVC]
    //   v.__set_size(n) [libc++]
    //   v._M_set_length(n) [libstdc++ 11]
  }
}
```

Introduction

- This talk is about P1072R10:

```
template <class CharT, class Traits, class Allocator>
template <typename Operation>
constexpr void std::basic_string<CharT, Traits, Allocator>::
  resize_and_overwrite(size_type count, Operation Op);
```

and hence esp.

```
std::string::resize_and_overwrite(count, op)
```

Introduction

- This talk is about P1072R10:

```
template <class CharT, class Traits, class Allocator>
template <typename Operation>
constexpr void std::basic_string<CharT, Traits, Allocator>::
  resize_and_overwrite(size_type count, Operation Op);
```

  and hence esp.

```
std::string::resize_and_overwrite(count, op)
```

- C++23

Introduction

- This talk is about P1072R10:

```
template <class CharT, class Traits, class Allocator>
template <typename Operation>
constexpr void std::basic_string<CharT, Traits, Allocator>::
  resize_and_overwrite(size_type count, Operation Op);
```

and hence esp.

```
std::string::resize_and_overwrite(count, op)
```

- C++23
- `*_overwrite` reminds of C++20:

```
std::make_unique_for_overwrite()
std::make_shared_for_overwrite()
```

Detour: make_unique_for_overwrite

-
  ```
  std::make_unique<T>(args...) ≙
  std::unique_ptr<T>(new T(args...))
  ```

DETOUR: MAKE_UNIQUE_FOR_OVERWRITE

-
  ```
  std::make_unique<T>(args...) ≙
  std::unique_ptr<T>(new T(args...))
  ```

-
  ```
  std::make_unique<T>() ≙
  std::unique_ptr<T>(new T())
  ```

Detour: make_unique_for_overwrite

-
  ```
  std::make_unique<T>(args...) ≙
  std::unique_ptr<T>(new T(args...))
  ```

-
  ```
  std::make_unique<T>() ≙
  std::unique_ptr<T>(new T())
  ```

-
  ```

  std::unique_ptr<T>(new T)
  ```

Detour: make_unique_for_overwrite

-
  ```
  std::make_unique<T>(args...) ≙
  std::unique_ptr<T>(new T(args...))
  ```

-
  ```
  std::make_unique<T>() ≙
  std::unique_ptr<T>(new T())
  ```

-
  ```
  std::make_unique_for_overwrite<T>() ≙
  std::unique_ptr<T>(new T)
  ```

Detour: make_unique_for_overwrite

-
  ```
  std::make_unique<T>(args...) ≙
  std::unique_ptr<T>(new T(args...))
  ```

-
  ```
  std::make_unique<T>() ≙
  std::unique_ptr<T>(new T())
  ```

  For non-class types (int/...): *Zero-initialized*

-
  ```
  std::make_unique_for_overwrite<T>() ≙
  std::unique_ptr<T>(new T)
  ```

  Default-initialized ⟶ non-class: *No initialization*

Detour: make_unique_for_overwrite

-
  ```
  std::make_unique<T>(args...) ≙
  std::unique_ptr<T>(new T(args...))
  ```

-
  ```
  std::make_unique<T>() ≙
  std::unique_ptr<T>(new T())
  ```
  For non-class types (int/...): *Zero-initialized*

-
  ```
  std::make_unique_for_overwrite<T>() ≙
  std::unique_ptr<T>(new T)
  ```
  Default-initialized ⟶ non-class: *No initialization*
- Read "indeterminate value": undefined behaviour!

Agenda

- Motivation
- Detour: std::make_unique_for_overwrite

Agenda

- Motivation
- Detour: std::make_unique_for_overwrite
- Example 1: Existing Solutions

Agenda

- Motivation
- Detour: std::make_unique_for_overwrite
- Example 1: Existing Solutions
- New Api: Basics

Agenda

- Motivation
- Detour: std::make_unique_for_overwrite
- Example 1: Existing Solutions
- New Api: Basics
- Example 1 w/ New Api

Agenda

- Motivation
- Detour: std::make_unique_for_overwrite
- Example 1: Existing Solutions
- New Api: Basics
- Example 1 w/ New Api
- Proposal Language

Agenda

- Motivation
- Detour: std::make_unique_for_overwrite
- Example 1: Existing Solutions
- New Api: Basics
- Example 1 w/ New Api
- Proposal Language
- std/cppreference Language

Agenda

- Motivation
- Detour: std::make_unique_for_overwrite
- Example 1: Existing Solutions
- New Api: Basics
- Example 1 w/ New Api
- Proposal Language
- std/cppreference Language
- Example 2

Agenda

- Motivation
- Detour: std::make_unique_for_overwrite
- Example 1: Existing Solutions
- New Api: Basics
- Example 1 w/ New Api
- Proposal Language
- std/cppreference Language
- Example 2
- Compiler/Library Support

Agenda

- Motivation
- Detour: std::make_unique_for_overwrite
- Example 1: Existing Solutions
- New Api: Basics
- Example 1 w/ New Api
- Proposal Language
- std/cppreference Language
- Example 2
- Compiler/Library Support
- Links

Example 1A

```cpp
std::string RepeatPattern(const std::string& pattern, size_t count)
{
  std::string ret;

  const auto step = pattern.size();
  // SUB-OPTIMAL: We memset step*count bytes only to overwrite them.
  ret.resize(step * count);
  for (size_t i = 0; i < count; i++) {
    // GOOD: No bookkeeping
    memcpy(ret.data() + i * step, pattern.data(), step);
  }

  return ret;
}
```

Example 1B

```
std::string RepeatPattern(const std::string& pattern, size_t count)
{
  std::string ret;

  ret.reserve(pattern.size() * count);
  for (size_t i = 0; i < count; i++) {
    // SUB-OPTIMAL:
    // * Writes 'count' nulls
    // * Updates size and checks for potential resize 'count' times
    // * Does not work for C APIs
    ret.append(pattern);
  }

  return ret;
}
```

New Api: Signature

```
template <typename Operation>
constexpr void resize_and_overwrite(
  size_type count,  // usually: std::size_t
  Operation Op
);
```

New Api: Signature

```
template <typename Operation>
constexpr void resize_and_overwrite(
  size_type count,  // usually: std::size_t
  Operation Op
);
```

with

```
size_t Operation(char* p, size_t n)
{
  [… write into *p, up to *(p + n) …]
  [… store number of actually written bytes into new_n …]
  // assert(new_n <= n);
  return new_n;
}
```

Example 1C

```
std::string RepeatPattern(const std::string& pattern, size_t count)
{
  std::string ret;

  const auto step = pattern.size();
  // GOOD: No initialization
  ret.resize_and_overwrite(step * count, [&](char* buf, size_t n) {
    for (size_t i = 0; i < count; i++) {
      // GOOD: No bookkeeping
      memcpy(buf + i * step, pattern.data(), step);
    }
    return step * count;
  });

  return ret;
}
```

Proposal language I

```
template <class Operation>
constexpr void resize_and_overwrite(size_type n, Operation op);
```

- Effects: Alters the value of `*this` as follows:

Proposal language I

```
template <class Operation>
constexpr void resize_and_overwrite(size_type n, Operation op);
```

- Effects: Alters the value of `*this` as follows:
    - If `n <= size()`, erases the last `size() - n` elements.

Proposal language I

```
template <class Operation>
constexpr void resize_and_overwrite(size_type n, Operation op);
```

- Effects: Alters the value of `*this` as follows:
  - If `n <= size()`, erases the last `size() - n` elements.
  - If `n > size()`, appends `n - size()` default-initialized elements.

Proposal language I

```
template <class Operation>
constexpr void resize_and_overwrite(size_type n, Operation op);
```

- Effects: Alters the value of `*this` as follows:
  - If `n <= size()`, erases the last `size() - n` elements.
  - If `n > size()`, appends `n - size()` default-initialized elements.
  - Invokes `erase(begin() + op(data(), n), end())`.

PROPOSAL LANGUAGE I

```
template <class Operation>
constexpr void resize_and_overwrite(size_type n, Operation op);
```

- Effects: Alters the value of `*this` as follows:
  - If `n <= size()`, erases the last `size() - n` elements.
  - If `n > size()`, appends `n - size()` default-initialized elements.
  - Invokes `erase(begin() + op(data(), n), end())`.
- Remarks:
  - If `op` throws the behavior is undefined.

PROPOSAL LANGUAGE I

```
template <class Operation>
constexpr void resize_and_overwrite(size_type n, Operation op);
```

- Effects: Alters the value of `*this` as follows:
  - If `n <= size()`, erases the last `size() - n` elements.
  - If `n > size()`, appends `n - size()` default-initialized elements.
  - Invokes `erase(begin() + op(data(), n), end())`.
- Remarks:
  - If `op` throws the behavior is undefined.
  - Let `o = size()` before the call to resize_and_overwrite.

PROPOSAL LANGUAGE I

```
template <class Operation>
constexpr void resize_and_overwrite(size_type n, Operation op);
```

- Effects: Alters the value of `*this` as follows:
  - If `n <= size()`, erases the last `size() - n` elements.
  - If `n > size()`, appends `n - size()` default-initialized elements.
  - Invokes `erase(begin() + op(data(), n), end())`.
- Remarks:
  - If `op` throws the behavior is undefined.
  - Let `o = size()` before the call to resize_and_overwrite.
  - Let `m = op(data(), n)`.

PROPOSAL LANGUAGE I

```
template <class Operation>
constexpr void resize_and_overwrite(size_type n, Operation op);
```

- Effects: Alters the value of `*this` as follows:
  - If `n <= size()`, erases the last `size() - n` elements.
  - If `n > size()`, appends `n - size()` default-initialized elements.
  - Invokes `erase(begin() + op(data(), n), end())`.
- Remarks:
  - If `op` throws the behavior is undefined.
  - Let `o = size()` before the call to resize_and_overwrite.
  - Let `m = op(data(), n)`.
  - `m <= n` otherwise the behavior is undefined.

PROPOSAL LANGUAGE I

```
template <class Operation>
constexpr void resize_and_overwrite(size_type n, Operation op);
```

- Effects: Alters the value of `*this` as follows:
  - If `n <= size()`, erases the last `size() - n` elements.
  - If `n > size()`, appends `n - size()` default-initialized elements.
  - Invokes `erase(begin() + op(data(), n), end())`.
- Remarks:
  - If `op` throws the behavior is undefined.
  - Let `o = size()` before the call to resize_and_overwrite.
  - Let `m = op(data(), n)`.
  - `m <= n` otherwise the behavior is undefined.
  - If `m > o`, `op` shall replace the values stored in the character array `[data() + o, data() + m)`.
    Until replaced, the values may be indeterminate
    [Note: `*(data() + o)` may not be `charT()`. - end note].

Proposal language II

- Remarks(cont'd):
  - `op` may write to `data() + n + 1`.
    Any value written will be replaced with `charT()` after op returns.
    [Note: This facilitiates interoperation with functions that write a trailing null. - end note]

Proposal language II

- Remarks(cont'd):
  - `op` may write to `data() + n + 1`.
    Any value written will be replaced with `charT()` after op returns.
    [Note: This facilitiates interoperation with functions that write a trailing null. - end note]
  - When `op` is called `*this` is in an unspecified state.
    `op` shall not bind or in any way access `*this`.

PROPOSAL LANGUAGE II

- Remarks(cont'd):
  - `op` may write to `data() + n + 1`.
    Any value written will be replaced with `charT()` after op returns.
    [Note: This facilitiates interoperation with functions that write a trailing null. - end note]
  - When `op` is called `*this` is in an unspecified state.
    `op` shall not bind or in any way access `*this`.
  - `op` shall not allow its first argument to be accessible after it returns.

std/cppreference Language I

std/cppreference Language I

- This function performs following steps:

std/cppreference Language I

- This function performs following steps:
  1. Obtains a contiguous storage that contains `count + 1` characters, and makes its first `k` characters equal to the first `k` characters of `*this`, where `k = min(count, this->size())`
     Let `p` denote the pointer to the first character in the storage.

STD/CPPREFERENCE LANGUAGE I

- This function performs following steps:
    1. Obtains a contiguous storage that contains `count + 1` characters, and makes its first `k` characters equal to the first `k` characters of `*this`, where `k = min(count, this->size())`

       Let `p` denote the pointer to the first character in the storage.
        - The equality is determined as if by checking `this->compare(0, k, p, k) == 0`.

STD/CPPREFERENCE LANGUAGE I

- This function performs following steps:
    1. Obtains a contiguous storage that contains `count + 1` characters, and makes its first `k` characters equal to the first `k` characters of `*this`, where `k = min(count, this->size())`

       Let `p` denote the pointer to the first character in the storage.
        - The equality is determined as if by checking `this->compare(0, k, p, k) == 0`.
        - The characters in `[p + k, p + count]` may have indeterminate values.

STD/CPPREFERENCE LANGUAGE I

- This function performs following steps:
  1. Obtains a contiguous storage that contains `count + 1` characters, and makes its first `k` characters equal to the first `k` characters of `*this`, where `k = min(count, this->size())`
     Let `p` denote the pointer to the first character in the storage.
     - The equality is determined as if by checking `this->compare(0, k, p, k) == 0`.
     - The characters in `[p + k, p + count]` may have indeterminate values.
  2. Evaluates `std::move(op)(p, count)`.
     Let `r` be the return value of `std::move(op)(p, count)`.

STD/CPPREFERENCE LANGUAGE I

- This function performs following steps:
  1. Obtains a contiguous storage that contains `count + 1` characters, and makes its first `k` characters equal to the first `k` characters of `*this`, where `k = min(count, this->size())`
     Let `p` denote the pointer to the first character in the storage.
     - The equality is determined as if by checking `this->compare(0, k, p, k) == 0`.
     - The characters in `[p + k, p + count]` may have indeterminate values.
  2. Evaluates `std::move(op)(p, count)`.
     Let `r` be the return value of `std::move(op)(p, count)`.
  3. Replaces the contents of `*this` with `[p, p + r)`,
     and set the length of `*this` to `r`.
     Invalidates all pointers and references to the range `[p, p + count]`.

std/cppreference Language II

- The program is ill-formed if `r` does not have an integer-like type.

STD/CPPREFERENCE LANGUAGE II

- The program is ill-formed if `r` does not have an integer-like type.
- The behavior is undefined if
  - `std::move(op)(p, count)` throws an exception

STD/CPPREFERENCE Language II

- The program is ill-formed if `r` does not have an integer-like type.
- The behavior is undefined if
  - `std::move(op)(p, count)` throws an exception
  - or modifies `p` or `count`,

std/cppreference Language II

- The program is ill-formed if `r` does not have an integer-like type.
- The behavior is undefined if
  - `std::move(op)(p, count)` throws an exception
  - or modifies `p` or `count`,
  - `r` is not in the range `[0, count]`,

std/cppreference Language II

- The program is ill-formed if `r` does not have an integer-like type.
- The behavior is undefined if
  - `std::move(op)(p, count)` throws an exception
  - or modifies `p` or `count`,
  - `r` is not in the range `[0, count]`,
  - or any character in range `[p, p + r)` has an indeterminate value.

STD/CPPREFERENCE LANGUAGE II

- The program is ill-formed if `r` does not have an integer-like type.
- The behavior is undefined if
  - `std::move(op)(p, count)` throws an exception
  - or modifies `p` or `count`,
  - `r` is not in the range `[0, count]`,
  - or any character in range `[p, p + r)` has an indeterminate value.

- **Implementations are recommended to avoid unnecessary copies and allocations** by, e.g., making `p` equal to the pointer to beginning of storage of characters allocated for `*this` after the call, which can be identical to the existing storage of `*this` if `count` is less than or equal to `this->capacity()`.

Example 2

```cpp
std::string CompressWrapper(std::string_view input) {
  std::string compressed;
  // Compute upper limit of compressed input (zlib: deflateBound())
  const size_t bound = compressBound(input.size());
  int err;

  // GOOD: No initialization
  compressed.resize_and_overwrite(bound, [&](char* buf, size_t n) {
    size_t out_size = n;
    err = compress(buf, &out_size, input.data(), input.size());
    return out_size;
  });
  if (err != OK) {
    throw ...some error...
  }
  return compressed;
}
```

Compiler/Library Support

- gcc/libstdc++: 12
- clang/libc++: 14 (Apple: 14.0.3 / Xcode 14.3)
- msvc stl: 19.31 / VS 2022 17.1

Compiler/Library Support

- gcc/libstdc++: 12
- clang/libc++: 14 (Apple: 14.0.3 / Xcode 14.3)
- msvc stl: 19.31 / VS 2022 17.1

-
  ```
  #if __cpp_lib_string_resize_and_overwrite >= 202110L
    [...]
  #else
    [...]
  #endif
  ```

Compiler/Library Support

- gcc/libstdc++: 12
- clang/libc++: 14   (Apple: 14.0.3 / Xcode 14.3)
- msvc stl: 19.31 / VS 2022 17.1

-
  ```
  #if __cpp_lib_string_resize_and_overwrite >= 202110L
    [...]
  #else
    [...]
  #endif
  ```
- boost/container has similar api e.g. for std::vector

Links, Questions?

- P1072R10
  https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/p1072r10.html
- https://en.cppreference.com/w/cpp/string/basic_string/resize_and_overwrite
- Daniel Frey:
  "Accessing Private Members the Right Way"
  https://cpp-aachen.github.io/archive/2023-11-29/access_private_members/AccessPrivateMembers.pdf
- Slides:
  https://smilingthax.github.io/slides/resize_overwrite/