Effective C++ Programming

NIE-EPC (v. 2021):

SMALL STRING OPTIMIZATION

String class — overview

- `std::string` — "*dynamic* string owner":
  - Has allocated buffer for some string length (number of its characters), which is called *capacity*.
  - Holds/owns/manages a string of some *size*, which is lower than or equal to the capacity.
- Simplified custom string-class implementation:

```
class String {
  char* data_;        // pointer to the owned string of characters
  size_t size_;       // its size (length)
  size_t capacity_;   // capacity (maximum length) of the allocated buffer
public:
  ...
};
```

- The capacity can grow to allow owning a string of characters of any length.
- *Grow of capacity* = storage "*reallocation*":
  - New buffer is allocated and the characters are copied from the old to the new buffer.

String class — default constructor

- *Default constructor* = ownership of *empty string*.
- *Empty string* = string with a single null character (`'\0'`).

```
class String {
  char* data_;
  size_t size_, capacity_;

public:
  String() : size_(0), capacity_(0), data_( new char[1] ) { data_[0] = '\0'; }
  ~String() { delete[] data_; }
};
```

- *Size* and *capacity* are counted in the number of string characters without the terminal null character, which is mandatory.
  - $\Rightarrow$ *capacity* = *buffer size* - 1 (and, *size* < *buffer size*).
- *Note:*
  - `new char[1]` causes so-called "*default-initialization*", which results in all char elements being initialized to unspecified values.
  - `new char[1]()` or `new char[1]{}` causes "*value-initialization*", which results in all char elements being initialized to zero, that is, to `'\0'` character.

```
String() : size_(0), capacity_(0), data_( new char[1]{} ) { }  // same effect
```

String class — converting constructor

- Constructor that creates ownership of the copy of the string of characters passed as an argument:

```
class String {
  char* data_;
  size_t size_, capacity_;

public:
  String(const char* arg) : size_(strlen(arg)), capacity_(size_),
                            data_(new char[capacity_ + 1]) {  // allocate buffer
    strcpy(data_, arg);                                       // copy characters
  }
  ...  // default constructor + destructor
};
```

- *Problem:*
  - Member variables are initialized in the order of their declarations.
    - ⇒ `data_` is initialized first;
    - ⇒ at the time, `capacity_` has not been initialized yet;
    - ⇒ `capacity_` has *unspecified value* and its reading causes *undefined behavior*.

String class — converting const. *(cont.)*

- *Possible solution I.:*
  - Reordering of member variable declarations:

```
class String {
  size_t size_, capacity_;
  char* data_;              // data_ is guaranteed to be initialized after capacity_
  ...
```

- *Possible solution II.:*
  - Assignment instead of initialization for `data_`:

```
class String {
  char* data_;
  size_t size_, capacity_;  // original order

public:
  String(const char* arg) : size_(strlen(arg)), capacity_(size_) {
    data_ = new char[capacity_ + 1];  // is guaranteed to happen after initialization...
    strcpy(data_, arg);               // ...of all subobjects (including capacity_)
  }
  ...
```

- In case of member variables of basic (non-class) types (as `char*`), there is effectively no difference between their *initialization* and *assignment*.
  - *Proof* — no difference in the generated machine code: [link].
- For class types, the difference may be significant.

strcpy *vs* memcpy

- Original (*left*) *vs* alternative (*right*) implementation:

```
String(const char* arg)
  : size_(strlen(arg)), capacity_(size_)
{
  data_ = new char[capacity_ + 1];
  strcpy(data_, arg);
}
```

```
String(const char* arg)
  : size_(strlen(arg)), capacity_(size_)
{
  data_ = new char[capacity_ + 1];
  memcpy(data_, arg, size_ + 1);
}
```

- *Difference:*
  - Both functions copies bytes (characters) between memory locations.
  - ⇒ Iterative process (*loop*), where the number of iterations is:
    1) known with `memcpy`,
    2) unknown with `strcpy`.
  - Ad 2) `strcpy` does not know, which iteration will be the last one ⇒ it must process all iterations sequentially.
  - Ad 1) `memcpy` can process iterations in parallel (enabled optimizations such as *loop unrolling*, using *vectorization/SIMD instructions*, etc.).
- Generally, memcpy may be faster, but it depends on the quality of its implementation.
- Also, the difference will likely be more significant for long strings.

String class — memory efficiency

- *Actual implementation:*

```
class String {
  char* data_;
  size_t size_, capacity_;

public:
  String() : size_(0), capacity_(0), data_( new char[1]{} ) { }

  String(const char* arg) : size_(strlen(arg)), capacity_(size_) {
    data_ = new char[capacity_ + 1];
    memcpy(data_, arg, size_ + 1);
  }

  ~String() { delete[] data_; }
};
```

- *Example use*:

```
String s("short");  // in some function
```

- Memory efficiency — *x86_64/Linux*:
  - Storage of s requires 24 bytes.
  - Dynamic allocation takes 32 bytes (16 bytes for *allocated chunk*, 16 bytes for *housekeeping data*; *see previous lectures*).
  - ⇒ To work with a string of 5 characters, 56 bytes are needed!
  - Only less than 10% of required memory contain *useful data*.

Short string optimization

- *Simple idea:*
  - In real-world programs, there are many *short strings* processed (such as *names of entities* in databases,…).
  - Storing such strings in *dynamically-allocated* storage is inefficient.
  - ⇒ Strings up to some length will be stored in the **included storage** of the string-class object itself.
- Such optimization technique is called *small/short string optimization* (SSO).
- Most straightforward implementation:
  - Adding a buffer member variable directly into the string-class itself:

```
class String {
  char* data_;
  size_t size_, capacity_;
  char buffer_[...];
public:
  ...
};
```

SSO — additional buffer

```
class String {
  char* data_;                // 8-bytes long, 8-byte aligned
  size_t size_, capacity_;  // ditto for both
  char buffer_[...];
  ...
```

- How to choose buffer size?
  - On a 64-bit architecture, alignment requirements for `String` class is 8.
  - We do not want any *wasted bytes* due to *padding*.
  - ⇒ `buffer_` size needs to be a multiple of 8.
- *Example:*

```
class String {
  char* data_;
  size_t size_, capacity_;
  char buffer_[16];
  ...
```

- `buffer_` now can contain a string of characters with up to the 15 characters + *terminal zero character* ⇒ its capacity is 15.
- ⇒ *Default constructor* — no need for dynamic allocation :)

```
String() : size_(0), capacity_(15), data_(buffer_) { buffer_[0] = '\0'; }
```

SSO — additional buffer *(cont.)*

```
String() : size_(0), capacity_(15), data_(buffer_) { buffer_[0] = '\0'; }
```

- How to recognize whether the owned string is *short* or *long*?
  - In case of *short* string, data_ points to buffer_,
  - Otherwise (*long* string), it points to the dynamically-allocated memory.

```
private:
  bool is_short() const { return data_ == buffer_; }  // private helper function
```

- In destructor, memory needs to be deallocated only if it has been dynamically-allocated before ⇒ only if the owned string is not *short*.

```
~String() { if (!is_short()) delete[] data_; }
```

- Finally, converting constructor needs to distinguish between *short* and *long* strings:

```
String(const char* arg) : size_(strlen(arg)) {
  if (size_ < 16) {
    capacity_ = 15;
    data_ = buffer_;
  } else {
    capacity_ = size_;
    data_ = new char[capacity_ + 1];
  }
  memcpy(data_, arg, size_ + 1);
}
```

SSO — additional buffer *(cont.)*

- Implementation with additional buffer:

```
class String {
  char* data_;
  size_t size_, capacity_;
  char buffer_[16];

  bool is_short() const { return data_ == buffer_; }

public:
  String() : size_(0), capacity_(15), data_(buffer_) { buffer_[0] = '\0'; }

  String(const char* arg) : size_(strlen(arg)) {
    if (size_ < 16) { capacity_ = 15;    data_ = buffer_;                  }
    else            { capacity_ = size_; data_ = new char[capacity_ + 1]; }
    memcpy(data_, arg, size_ + 1);
  }

  ~String() { if (!is_short()) delete[] data_; }
};
```

- Memory efficiency for "short" string:

```
String s("short");
```

  - Storage for s now requires 24 + 16 = 40 bytes (originally, it was 24).
  - No dynamic memory allocation is required.
  - ⇒ To work with a string of 5 characters, 40 bytes are needed (originally, it was 56) ⇒ memory "efficiency" 12.5%.
  - Maximum *short* string has 15 characters ⇒ efficiency 37.5%.

SSO — additional buffer — *union*

- Let us add functions for getting owned string size and allocated buffer capacity (as `std::string` provide):

```
class String {
  ...
public:
  ...
  size_t capacity() const { return capacity_; }
  size_t size() const { return size_; }
};
```

- *Observation*:
  - If the owned string is *short*, capacity is fixed (15).
  - ⇒ It does not need to be explicitly stored; it can be derived instead:

```
size_t capacity() const { return is_short() ? 15 : capacity_; }
```

- *Outcome*:
  - If the string is *short*, `capacity_` member variable is unused.
  - If the string is *long*, `buffer_` member variable is unused.
  - ⇒ They can be stored in the same storage.

SSO — additional buffer — *union (cont.)*

- Storage "sharing" = *union* types (*introduced in "UB" lecture*).
- **Union** = class type where all member variables (*subobjects*) uses/shares the same storage.
- Original (*left*) and new (*right*) `String` implementation:

```
class String {
  char* data_;
  size_t size_;
  size_t capacity_;
  char buffer[16];
  ...
```

```
std::cout << sizeof(String);  // "40"
```

```
class String {
  char* data_;
  size_t size_;

  union {
    size_t capacity_;
    char buffer[16];
  };
  ...
```

```
std::cout << sizeof(String);  // "32"
```

- Unions have some limitations that makes them hard to use especially with class-type members.
- The C++ Standard library provides "*safe*" union — `std::variant`.
  - Price for safety are larger storage requirements due to the need for "housekeeping" data.
  - ⇒ In our case, we will stick to "ordinary" union.

SSO — additional buffer — *union (cont.)*

- Implementation with additional buffer and union:

```
class String {
  char* data_;
  size_t size_;
  union { size_t capacity_; char buffer_[16]; }  // storage sharing

  bool is_short() const { return data_ == buffer_; }

public:
  String() : size_(0), capacity_(15), data_(buffer_) { buffer_[0] = '\0'; }

  String(const char* arg) : size_(strlen(arg)) {
    if (size_ < 16) { /* nothing */      data_ = buffer_;                 }
    else            { capacity_ = size_; data_ = new char[capacity_ + 1]; }
    memcpy(data_, arg, size_ + 1);
  }

  ~String() { if (!is_short()) delete[] data_; }

  size_t size() const { return size_; }
  size_t capacity() const { return is_short() ? 15 : capacity_; }
};
```

- Memory efficiency for “short” string:

```
String s("short");
```

  - Storage for s now requires 16 + 16 = 32 bytes (without union, it was 40).
  - ⇒ To work with a string of 5 characters, 32 bytes are needed (without union, it was 40) ⇒ memory “efficiency” 15.6%.
  - Maximum *short* string has 15 characters ⇒ efficiency 46.9%.

SSO — additional buffer — *portability*

- Owned string accessor:

```
const char* data() const { return data_; }
```

- Recall (*see lecture about undefined behavior*) that:
  - In C++, only *active* union member may be accessed.
  - *Active* = last set/assigned/written into.
- In our case, this requirement is satisfied.
- When the string is *short*:
  - characters are written into `buffer_` (by `memcpy`), which makes it active;
  - `capacity_` is not used.
- When the string is *long*:
  - `capacity_` is set, which makes it active;
  - `buffer_` is not used.
- ⇒ Our implementation is portable and does not need any non-standard language extension regarding to unions.

SSO — additional buffer — *libstdc++/MSTL*

- SSO implemented with the additional buffer is implemented by:
  - *GNU libstdc++* and *Microsoft STL* standard library implementations.

```
std::string s;
std::cout << sizeof(s);      // prints out "32" with libstdc++/MSTL on x86_64
std::cout << s.capacity();   // prints out "15" with libstdc++/MSTL on x86_64
```

- Implementation in *libstdc++* (*include/bits/basic_string.h*):

```
enum { _S_local_capacity = 15 / sizeof(_CharT) };

union
{
  _CharT    _M_local_buf[_S_local_capacity + 1];
  size_type _M_allocated_capacity;
};
```

- Likely *non-persistent* [link].
- `std::basic_string` is a template parametrized by character type (`_CharT`).
- `std::string` = instance of `std::basic_string` where character type `_CharT` = `char`.
- ⇒ `sizeof(_Char)` is 1 ⇒ "short capacity" is 15, buffer size is 16.

SSO — aliased buffer

- Is it possible even more increase memory efficiency for short strings?
- Let's get back to the original non-SSO `String` implementation:

```
class String {
  char* data_;
  size_t size_, capacity_;
  ...
```

- *Assumption:* 64-bit architecture.
- ⇒ Each member variable require 8-byte storage.
- ⇒ In total, they require 24 bytes.
- Would it be possible to share storage of all of them with a buffer for short strings, such as…?

```
class String {
  union {
    struct { char* data; size_t size, capacity; } long_;  // data for long strings
    char buffer_[24];                                      // buffer for short strings
  };
  ...
```

- Objects that share storage are called to be "*aliased*".
- ⇒ We call this approach "aliased buffer" case.

SSO — aliased buffer — *short vs long*

- A lot of problems need to be resolved.
- *First problem* — how to distinguish between short and long strings?
  - The information of whether a *short* or *long* string is owned needs to be stored somewhere.
  - ⇒ We need to reserve some byte of the storage.
- We will reserve the its last byte:

```
class String {
  union {
    struct { char* data; size_t size, capacity; } long_;   // data for long strings
    struct { char buffer[23]; bool flag; } short_;         // data for short strings
  };
  ...
```

- `sizeof(bool)` is not guaranteed to be 1.
- This is guaranteed of (unsigned) char type:

```
 struct { char buffer[23]; unsigned char flag; } short_;
```

- Now, we can implement `is_short()` as follows:

```
bool is_short() const { return short_.flag; }  // 1 => short string
```

SSO — aliased buffer — *portability*

- *Second problem*:
  - The information about *short/long* string is read from `short_` member of the union.
  - However, until we read it, we don't know which member of the union is active.
  - If `short_` is not active, reading `short_.flag` results in undefined behavior according to the C++ standard.
  - $\Rightarrow$ This SSO solution requires a C++ implementation that supports reading non-active union members (such as *GCC* or *Clang*; basically, all mainstream implementations do support it).
  - *Portable alternative?*

```
class String {
  union {
    struct { char* data; size_t size, capacity; } long_;   // data for long strings
    char buffer[24];                                       // buffer for short strings
  };
  unsigned char flag_;  // short/long-string flag outside of union
  ...
```

  - We are trying to maximize memory efficiency.
  - This solution would add 8 bytes (1 byte flag, 7 bytes *wasted* padding).

SSO — aliased buffer — *accessor*

```
class String {
  union {
    struct { char* data; size_t size, capacity; } long_;
    struct { char buffer[23]; unsigned char flag; } short_;
  };

  bool is_short() const { return short_.flag; }

public:
  ...
};
```

- *Third problem* — how to get access to the owned string?
  - Pointer `long_.data` shares storage with `short_.buffer`.
  - ⇒ There is no explicit *pointer* member variable to the string if it is *short*.
  - However, we know that in such a case, the string is in `short_.buffer`.
  - Otherwise — if owned string is *long* — it is pointed to by `long_.data`.

```
const char* data() const { return is_short() ? short_.buffer : long_.data; }
```

- *Helper* functions (*to-be-used-later*):

```
private:
  char* ptr() { return is_short() ? short_.buffer : long_.data; }
  const char* ptr() const { return is_short() ? short_.buffer : long_.data; }

public:
  const char* data() const { return ptr(); }
```

SSO — aliased buffer — *string size*

```
class String {
  union {
    struct { char* data; size_t size, capacity; } long_;
    struct { char buffer[23]; unsigned char flag; } short_;
  };
  bool is_short() const { return short_.flag; }
  char* ptr() { return is_short() ? short_.buffer : long_.data; }
  const char* ptr() const { return is_short() ? short_.buffer : long_.data; }
public:
  ...
  const char* data() const { return ptr(); }
};
```

- *Fourth problem* — how to resolve string size?
  - If the owned string is *long*, its size is in `long_.size`.
  - Variable `long_.size` shares storage with `short_.buffer`.
  - ⇒ There is no explicit *size* member variable if the string is *short*.
  - *Solution?*

```
size_t size() const { return is_short() ? strlen(short_.buffer) : long_.size; }
```

- We are trying to "mimic" `std::string`.
- `std::string::size()` requires *constant* time complexity.
- In our case, time complexity of `String::size()` is *linear*.

SSO — aliased buffer — *string size (cont.)*

- The only option on how to provide *short* string size in $O(1)$ time is to explicitly store it somewhere.
- In case of *short* strings, all bytes are occupied:
  - 23 bytes by buffer `short_.buffer`,
  - 1 byte by *short/long* string flag `short_.flag`.
- However, for the *short/long* flag, we in fact need only a single bit.
  - We will use the *least-significant-bit* (LSb).
- ⇒ Remaining 7 bits may be used for storing short-string size (which cannot have more than 22 characters).

```
class String {
  union {
    struct { char* data; size_t size, capacity; } long_;
    struct { char buffer[23]; unsigned char size_flag; } short_;
  };
  bool is_short() const { return short_.size_flag & 0x01; }
  size_t short_size() const { return short_.size_flag >> 1; }
  void short_size(size_t n) { short_.size_flag = n << 1 | 1; }
  ...
public:
  size_t size() const { return is_short() ? short_size() : long_.size; }
```

SSO — aliased buffer — *capacity*

```
class String {
  union {
    struct { char* data; size_t size, capacity; } long_;
    struct { char buffer[23]; unsigned char size_flag; } short_;
  };
  bool is_short() const { return short_.size_flag & 0x01; }
  size_t short_size() const { return short_.size_flag >> 1; }
  void short_size(size_t n) { short_.size_flag = n << 1 | 1; }
  char* ptr() { return is_short() ? short_.buffer : long_.data; }
  const char* ptr() const { return is_short() ? short_.buffer : long_.data; }
public:
  size_t size() const { return is_short() ? short_size() : long_.size; }
  const char* data() const { return ptr(); }
  ...
```

- *Fifth problem* — how to resolve capacity?
  - If the owned string is *short*, (buffer) capacity is 22 (buffer size is 23 but *terminal null character* needs to be stored in it as well).
  - If the owned string is *long*, is allocated capacity in `long_.capacity`?
  - Variable `long_.capacity` shares storage with both `short_.buffer` and `short_.size_flag`.
  - In case of *long* string, the LSb of `short_.size_flag` is zero.
  - ⇒ The same bit in `long_.capacity` is zero as well!

SSO — aliased buffer — *capacity (cont.)*

```
class String {
  union {
    struct { char* data; size_t size, capacity; } long_;
    struct { char buffer[23]; unsigned char size_flag; } short_;
  };
  ...
```

- ⇒ Capacity for *long* strings cannot be set to any value.
  - The byte that shares storage with `short_.size_flag` needs to have its LSb zero.
  - It is a byte of storage of `long_.capacity` with highest address.
- *Which byte is that?*
  - The answer depends on the *endianness* of the architecture.
  - *Big endian* —the *least-significant byte* (LSB) have the highest address.
  - *Little endian* — the *most-significant byte* (MSB) have the highest address.
- *Assumption* — *little endian* (*x86, x86_64,…*):

```
size_t capacity() const { return is_short() ? 22 : long_.capacity; }
```

SSO — aliased buffer — *capacity (cont.)*

```
class String {
  union {
    struct { char* data; size_t size, capacity; } long_;
    struct { char buffer[23]; unsigned char size_flag; } short_;
  };
  ...
public:
  ...
  size_t capacity() const { return is_short() ? 22 : long_.capacity; }
  ...
```

- *Consequence:*
  - `short_.size_flag` shares storage with the MSB of `long_.capacity`.
  - ⇒ LSb of MSB of `long_.capacity` must be zero.
  - ⇒ Practically, whole MSB of `long_.capacity` must be zero.
  - This limits the capacity for strings owned by String class owner to $2^{56} - 1$ characters, which corresponds to 64 PB.
- *Alternative option* — reordering of variables:

```
class String {
  union {
    struct { size_t capacity, size; char* data; } long_;          // capacity is first
    struct { unsigned char size_flag; char buffer[23]; } short_;  // size_flag is first
  };
  ...
```

SSO — aliased buffer — *capacity (cont.)*

```
class String {
  union {
    struct { size_t capacity, size; char* data; } long_;          // capacity is first
    struct { unsigned char size_flag; char buffer[23]; } short_;  // size_flag is first
  };
  ...
```

- Now, `short_.size_flag` shares storage with **LSB** of `long_.capacity`.
  - ⇒ LSb of LSB of `long_.capacity` must be zero.
  - ⇒ Capacity for *long* strings is *even*; no other restrictions.
- *Alternative alternative:*
  - *Even* capacity requires to allocate *odd* number of bytes.
  - Generally, it is more efficient to dynamically-allocate *even* number of bytes.
  - This corresponds with the *odd* capacity for long strings.
  - ⇒ We need LSb of `short_.size_flag` to be set (1) for *long* strings.

```
  bool is_short() const { return !(short_.size_flag & 0x01); }  // LSb 1 => long string
  ...
  void short_size(size_t n) { short_.size_flag = n << 1 /* | 1 */ ; }
```

SSO — aliased buffer — *libc++*

```
class String {
  union {
    struct { size_t capacity, size; char* data; } long_;          // capacity is first
    struct { unsigned char size_flag; char buffer[23]; } short_;  // size_flag is first
  };
  ...
```

```
  bool is_short() const { return !(short_.size_flag & 0x01); }  // LSb 1 => long string
```

- This approach is used in *LLVM libc++* standard library implementation:

```
std::string s;
std::cout << sizeof(s);     // prints out "24" with libc++ on x86_64
std::cout << s.capacity();  // prints out "22" with libc++ on x86_64
```

- Likely *non-persistent* [link]; file *include/string.*
- *Odd* capacity for *long* strings:

```
std::string s(24, 'A');     // owned string have 24 characters
std::cout << s.capacity();  // printed out "31" in tested case with libc++ on x86_64
```

- Capacity 31 ⇒ dynamically-allocated 32 bytes on the heap.
- Allocations being multiples of 16 are efficient (no alignment-enforced padding).

SSO — aliased buffer — *const/destructors*

- *Default constructor:*

```
String() : short_.size_flag(0) { short_.buffer_[0] = '\0'; }
```

- *Converting constructor:*

```
String(const char* arg) {
  size_t size = strlen(arg);
  if (size < 23) {  // short-string case:
    short_.size_flag = 0;  // clear LSb
    short_size(size);      // store size into higher 7 bits of short_.size_flag
  } else {          // long-string case:
    long_.capacity = size;
    if ((long_.capacity & 0x01) == 0) long_.capacity++;  // make capacity odd if necessary
    long_.size = size;
    long_.data = new char[long_.capacity + 1];
  }
  memcpy(ptr(), arg, size + 1);
}
```

- *Destructor:*

```
~String() { if (!is_short()) delete[] long_.data; }
```

SSO — aliased buffer — *portability*

- *Portability issues* — our implementation is now not portable.
- It works only under following assumptions:

1) Storage for the `long_` structure requires 24 bytes (64-bit architecture).

```
union {
  struct { size_t capacity, size; char* data; } long_;
  struct { unsigned char size_flag; char buffer[23]; } short_;
};
```

2) C++ implementation supports access of inactive union member.

```
bool is_short() const { return !(short_.size_flag & 0x01); }  // accessed even if long
```

3) System architecture endianness is *little-endian*.
   - ⇒ It forces `long_.capacity` to have the bit number `0` (LSb) set.
   - With *big-endian*, this approach would force the capacity have the bit number 56 set ⇒ *insanely large number*.

SSO — aliased buffer — *magic numbers*

- Resolving portability — ad 1):
  - There are a lot of "*magic numbers*" in our source code.
  - Using such *magic numbers* is almost always a sign of a bad practice.

```
class String {
  union {
    struct { size_t capacity, size; char* data; } long_;
    struct { unsigned char size_flag; char buffer[23]; }  short_;
  };
  bool is_short() const { return !(short_.size_flag & 0x01); }
  size_t short_size() const { return short_.size_flag >> 1; }
  void short_size(size_t n) { short_.size_flag = n << 1; }
  char* ptr() { return is_short() ? short_.buffer : long_.data; }
  const char* ptr() const { return is_short() ? short_.buffer : long_.data; }
public:
  String() : short_.size_flag(0) { short_.buffer_[0] = '\0'; }
  String(const char* arg) {
    size_t size = strlen(arg);
    if (size < 23) { short_.size_flag = 0; short_size(size); }
    else {
      long_.capacity = size; if ((long_.capacity & 0x01) == 0) long_.capacity++;
      long_.size = size; long_.data = new char[long_.capacity + 1];
    }
    memcpy(ptr(), arg, size + 1);
  }
  ~String() { if (!is_short()) delete[] long_.data; }
  size_t capacity() const { return is_short() ? 22 : long_.capacity; }
  size_t size() const { return is_short() ? short_size() : long_.size; }
  const char* data() const { return ptr(); }
};
```

SSO — aliased buffer — *magic nums (cont.)*

- *Portable solution:*

```
class String {
  struct long_t { size_t capacity, size; char* data; };
  static const size_t short_cap_ = sizeof(long_t) - 2;
  struct short_t { unsigned char size_flag; char buffer[short_cap_ + 1]; }

  union {
    long_t long_;
    short_t short_;
  };

  ...

public:

  String(const char* arg) {
    size_t size = strlen(arg);
    if (size <= short_cap_) { short_.size_flag = 0; short_size(size); }
    else {
      long_.capacity = size; if ((long_.capacity & 0x01) == 0) long_.capacity++;
      long_.size = size; long_.data = new char[long_.capacity + 1];
    }
    memcpy(ptr(), arg, size + 1);
  }

  size_t capacity() const { return is_short() ? short_cap_ : long_.capacity; }
  ...
};
```

- ⇒ No magic numbers regarding capacity for *short* strings.

SSO — aliased buffer — *portability (cont.)*

- Resolving portability — ad 2):
  - Cannot be avoided (without additional storage costs).
  - ⇒ This SSO solution is generally not portable.
  - ⇒ Can be used only with C++ implementations that support reading inactive union member (most implementations do support this).
- Resolving portability — ad 3):
  - On *big endian*, we could switch back to the original variable order:

```
struct long_t { char* data; size_t size, capacity; };
static const size_t short_cap_ = sizeof(long_t) - 2;
struct short_t { char buffer[short_cap_ + 1]; unsigned char size_flag; }
```

  - Here, again, `short_.size_flag` shares storage with LSB of `long_.capacity` (which is what we need).
  - Typical solution — *conditional compilation*:

```
#ifdef LITTLE_ENDIAN
  struct long_t { size_t capacity, size; char* data; };
  ...
#else
  struct long_t { char* data; size_t size, capacity; };
  ...
#endif
```

SSO — aliased buffer — *comparison*

- Memory efficiency for "short" string:

```
String s("short");
```

  - Storage for s now requires 24 bytes.
  - ⇒ To work with a string of 5 characters, 24 bytes are needed.
    ⇒ Memory "efficiency" 20.8%.
  - Maximum *short* string has 22 characters ⇒ efficiency 91.7%.
- *Comparison of memory efficiency:*
  1) *String without SSO* always dynamically allocates memory.
     - The *worst-case efficiency* (1 character-string) is **1.8%**.
  2) *String with additional buffer and union* — does not allocate for up to 15 characters.
     - The *worst-case efficiency* is **3.1%**.
     - The *best-case efficiency* for *short* strings is **46.9%**.
  3) *String with aliased buffer* — does not allocate for up to 22 characters.
     - The *worst-case efficiency* is **4.2%**.
     - The *best-case efficiency* for *short* strings is **91.7%**.

SSO — aliased buffer — *comparison (cont.)*

- Why *libstdc++* and *MSTL* do not use the same approach as *libc++*?
- Drawbacks of "*aliased buffer*" solution:
  1) It is much (much) more complicated (*right*):

```
class String {
  char* data_;
  size_t size_;
  union { size_t capacity_; char buffer_[16]; }
  bool is_short() const { return data_ == buffer_; }
public:
  String() : size_(0), capacity_(15), data_(buffer_) { buffer_[0] = '\0'; }
  String(const char* arg) : size_(strlen(arg)) {
    if (size_ < 16) {                   data_ = buffer_;                    }
    else            { capacity_ = size_; data_ = new char[capacity_ + 1]; }
    memcpy(data_, arg, size_ + 1);
  }
  ~String() { if (!is_short()) delete[] data_; }
  size_t size() const { return size_; }
  size_t capacity() const { return is_short() ? 15 : capacity_; }
};
```

```
class String {
#ifdef LITTLE_ENDIAN
  struct long_t { size_t capacity, size; char* data; };
  static const size_t short_cap_ = sizeof(long_t) - 2;
  struct short_t { unsigned char size_flag; char buffer[short_cap_ + 1]; }
#else
  struct long_t { char* data; size_t size, capacity; };
  static const size_t short_cap_ = sizeof(long_t) - 2;
  struct short_t { char buffer[short_cap_ + 1]; unsigned char size_flag; }
#endif
  union { long_t long_; short_t short_; };
  bool is_short() const { return !(short_.size_flag & 0x01); }
  size_t short_size() const { return short_.size_flag >> 1; }
  void short_size(size_t n) { short_.size_flag = n << 1; }
  char* ptr() { return is_short() ? short_.buffer : long_.data; }
  const char* ptr() const { return is_short() ? short_.buffer : long_.data; }
public:
  String() : short_.size_flag(0) { short_.buffer_[0] = '\0'; }
  String(const char* arg) {
    size_t size = strlen(arg);
    if (size <= short_cap_) { short_.size_flag = 0; short_size(size); }
    else {
      long_.capacity = size; if ((long_.capacity & 0x01) == 0) long_.capacity++;
      long_.size = size; long_.data = new char[long_.capacity + 1];
    }
    memcpy(ptr(), arg, size + 1);
  }
  ~String() { if (!is_short()) delete[] long_.data; }
  size_t capacity() const { return is_short() ? short_cap_ : long_.capacity; }
  size_t size() const { return is_short() ? short_size() : long_.size; }
  const char* data() const { return ptr(); }
};
```

  2) It is not fully portable.
  3) It imposes runtime overhead into each access to the owned string, as well as to other operations.

- *Additional buffer* — there is an explicit direct pointer to stored string.

```
const char* data() const { return data_; }
```

- *Aliased buffer* — pointer to stored string needs to be derived:

```
const char* ptr() const { return is_short() ? short_.buffer : long_.data; }
```

Branching and branch prediction

```
const char* data() const { return data_; }  // (1)
```

- *...versus:*

```
const char* ptr() const { return is_short() ? short_.buffer : long_.data; }  // (2)
```

- What is so "*wrong*" about (2)?
  - Generally, *branching = performance penalty*.
- Modern processors support:
  - *Pipelining* = processing multiple instructions at once.
  - *Out-of-order execution* = processing instructions in different order (than written in the program machine code).

- Branching hinders both since the processor does not know which way will the program continue to run.
- Reduction of performance penalty — *branch prediction*:
  - Processors try to guess which branch will take place and will consider their instructions to be executed.

Branching and branch prediction *(cont.)*

- Branch prediction possible outcomes:

1) *Correct guess:*
   - Almost no performance penalty.
2) *Incorrect guess:*
   - Effects of *pipeline-/out-of-order*-processed instructions needs to be discarded.
   - The instructions from mis-predicted branch are started to be processed from the beginning.
   - $\Rightarrow$ Significant performance penalty (in terms of CPU cycles) with respect to case 1).

- Interesting Stack Overflow post about branch prediction:
  - https://stackoverflow.com/q/11227809/580083
  - It is the most-voted question with the [C++] tag.

Branching and branch prediction *(cont.)*

- Since C++20, we can provide a *hint* for the compiler to indicate which branch is more-likely.
- On some architectures, this allows generating machine code that:
  - instructs the branch-prediction processor unit which branch will likely take place,
  - makes the execution of the more-likely branch more efficient.
- *Example* — source code (*left*), machine code (*right*):

```cpp
void true_path();
void false_path();

void f(bool cond) {
  if (cond) [[likely]] true_path();
  else                 false_path();
}
```

```asm
f(bool):
  test  dil, dil
  je    .L2
  jmp   true_path()
.L2:
  jmp   false_path()
```

```cpp
void f(bool cond) {
  if (cond) [[unlikely]] true_path();
  else                   false_path();
}
```

```asm
f(bool):
  test  dil, dil
  jne   .L4
  jmp   false_path()
.L4:
  jmp   true_path()
```

Branching and branch prediction *(cont.)*

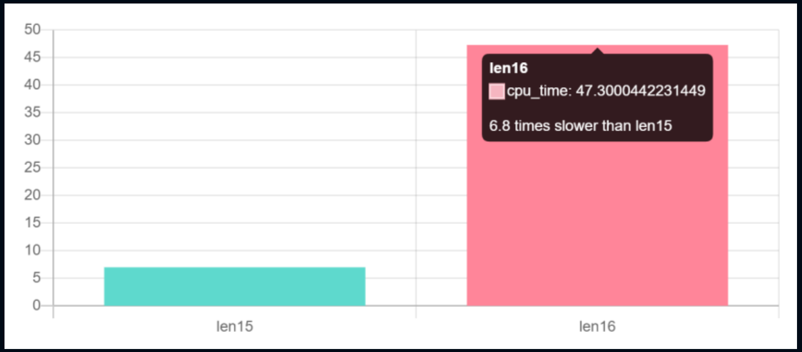
- Source code entities `[[likely]]` and `[[unlikely]]` are so-called C++ *attributes*.
- Their usage makes sense only in performance-critical code paths where coder can predict branching.
- What about our `String` class with aliased buffer?

```
const char* ptr() const { return is_short() ? short_.buffer : long_.data; }
```

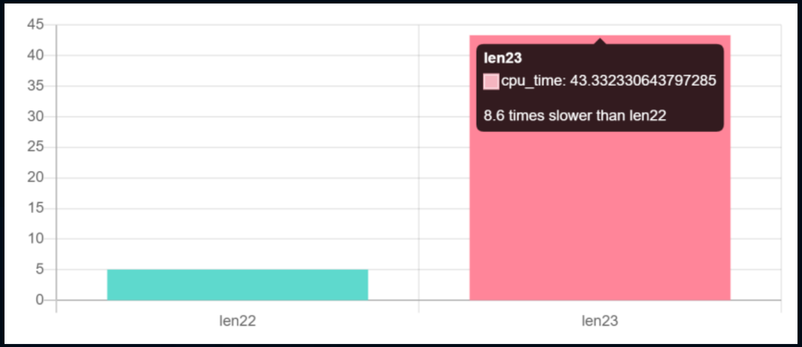
  - Creating string-owner objects may be generally performance-critical (e.g., in programs that process many strings objects such as database engines).
  - However, when implementing `String` class, there is absolutely no knowledge or even guess of whether in programs the owned string will be *short* or *long*.
- ⇒ It does not make sense to use these attributes here.

SSO — benchmark

- Measuring construction + destruction time for *short* and *long* strings with `std::string` and *libstdc++*:
  - *Short* string having 15 characters.
  - *Long* string having 16 characters.
  - *Results* — *short-string* case was **6.8× faster**.
  - *Quick C++ Benchmark* [link].



- The same with *libc++*:
  - *Short* string having 22 characters.
  - *Long* string having 23 characters.
  - *Results* — *short-string* case was **8.6× faster**.
  - *Quick C++ Benchmark* [link].



- ⇒ The benefits of SSO for *short* strings is significant.

SSO — the power of C++

- Implementation of SSO illustrates one of the major C++ strengths:
  - When implementing a string owner class, we use low-level abstraction mechanisms that allows us to precisely control what happens on the system level.
    - This wouldn't be possible without *pointers* (many languages don't have them), *controlling access to individual bits in memory*, *taking care about size and alignment requirements of objects storage*, *aliasing objects*, etc.
  - At the same time, application of SSO if fully transparent for class users.
    - They can write programs at a high level of abstraction and just use the string class without any knowledge of SSO.
  - Whether SSO is or is not internally applied does not change the way how the class is used (its API is preserved).
- *Note:*
  - Our implementation just showed a basic ideas of SSO.
    - It is by far not a complete string class; many functionalities are missing (*copy/move semantics*, *adding/removing content*, etc.).
    - In the current form, it is even *flawed*, since auto-generated copy constructor and copy assignment operator would now work incorrectly (for example, they would create shallow copies for *long* strings).

Bonus — `folly::fbstring`

- With *aliased buffer*, 24 bytes are available for a *short* string, which must include:
  1) *terminal null character* (first "housekeeping" byte),
  2) information about *string size* + information *that the string is short* (second "housekeeping" byte).
- ⇒ 22 bytes can be used for "effective" string characters.
- *Interesting idea:*
  - What if we managed to encode 2) in such a way, that the corresponding byte was zero for a string with 23 characters?
  - ⇒ Then, we could effectively "merge" both bytes into one, which would serve for both 1) and 2) at the same time.
- *How to do this?*
  - We need this byte to have the highest address.
  - Its LSb need to be zero for *short* strings.
  - Its remaining 7 bits must be zero for strings with 23 characters.

Bonus — folly::fbstring *(cont.)*

- The solution is to store — into those 7 bits — not the *size* of the short string itself, but x instead, where $x = 23 - \mathit{size}$.
- *Consequence*:
  - In case of a string with 23 characters, all the bits are zero and the highest byte case serve as the *terminal null character* at the same time.
  - Capacity of *short* strings is increased from 22 to 23.

```
class String {
  union {
    struct { char* data; size_t size, capacity; } long_;
    char buffer_[24];
  };
  bool is_short() const { return !(buffer_[23] & 0x01); }
  size_t short_size() const { return 23 - (buffer_[23] >> 1); }
  void short_size(size_t n) { buffer_[23] = 23 - n << 1; }
  char* ptr() { return is_short() ? buffer_ : long_.data; }
  const char* ptr() const { return is_short() ? buffer_ : long_.data; }
  ...
public:
  size_t capacity() const { return is_short() ? 23 : long_.capacity; }
  size_t size() const { return is_short() ? short_size() : long_.size; }
  ...
```

- *Note* — `size()` has still constant time complexity.

Bonus — folly::fbstring *(cont.)*

- This "*extreme*" *approach* is used by `folly::fbstring`.
- *Facebook Folly library* — provides "*a variety of core library components designed with efficiency in mind, which complements offerings such as Boost and `std`*".
- `folly::fbstring` — *string-class* similar to `std::string`.
  - Function for setting small size (similar to our `short_size`):

```
void setSmallSize(size_t s) {
  ...
  small_[maxSmallSize] = char((maxSmallSize - s) << shift);
  ...
```

- *Discussion:*
  - Many times — in real world — we can't have only pros without any cons.
  - *Example* — additional buffer (*faster*) *vs* aliased buffer (*more memory efficient*).
  - Costs of `fbstring` approach?
    - With our (*libc++*) *aliased buffer*, empty string = all storage bits/bytes are zero.
    - With `fbstring`, one byte needs to be nonzero (`23 - 0 << 1`).
    - This may add some overhead (for *move semantics*, *swapping*, etc.).