

ustring

A Modern Alternative to `std::string`

Alan Talbot

14 May 2012

The Program

- Lecture
 - The Problem
 - The Solution
 - The Objections
 - The Proposal
- Feedback
 - Wherein everyone gets to offer suggestions, ideas, theories, criticisms, and ridicule

The Problem

- Strings
 - Almost every program deals with strings, and a large and important class of programs require very efficient string processing
 - Some programs have special string handling requirements
 - In our polyglot global world, string handling has become all the more challenging
- The venerable `std::string`
 - Has served us well for many years
 - Has some significant fundamental limitations
 - Has some rather annoying quirks
 - Lacks features that are common in other libraries and languages
 - Is not sufficiently global
 - Does not take advantage of modern C++ design (especially C++11)
- Therefore
 - Most significant programs need more than one string type
 - I typically use three in Windows programs: char arrays and pointers, `std::string`, and `CString`
 - This leads to kind of a mess

Inflexible Memory Model

- Local memory usage is implementation defined
 - Small string optimization is typically present
 - No size vs. speed control
 - Inefficient for small strings where memory is tight
- Growth behavior is implementation defined
 - Typical growth is exponential
 - Inefficient for small strings where memory is tight because of air
 - Inefficient or even impossible for strings that are large in relation to addressable memory size
- Not byte compatible with C-style char arrays
 - C-style arrays are sometimes what you have, but then copies are required
 - C-style arrays have advantages in some situations
- Cannot be used if a local-only (no heap) allocation is required
 - Cannot be efficiently embedded in composite structures where size optimization is desired

Interaction With C Strings

- Interaction with char arrays
 - To construct a `std::string` from a char array requires a copy
 - To get a char array back out of a `std::string` requires a copy
- Interaction with `char*`s
 - If you want to support both native strings and `std::strings`, you need (at least) two overloads:
 - `void foo(const char* str);`
 - `void foo(const string& str);`
 - If you have only the first, you can't use natural syntax—you have to use `std::string::c_str()`, and hope that it's free
 - If you have only the second, you'll get an extra construction, copy and destruction
 - Oh, by the way, all the Standard Library string manipulation routines are in C, so they take `char*` only

Functionality Limitations

- Many common operations are not directly supported
 - Trim
 - Make upper/lower
 - Case-insensitive compare
 - Token extraction
 - Format (a la printf)
- Most string manipulations are handled by the CRT
 - But the CRT is not well supported
 - Functions are not composable, and they use conflicting metaphors
- Unicode
 - `std::string` does not support Unicode
 - `std::wstring` does not really support Unicode
 - No interoperability between these variants

Functionality Limitations

- Building strings
 - Building strings from other types (numbers, etc.) is not supported directly
 - Using `std::stringstream` is extra code and often inefficient
 - Using `to_string` is inefficient and inflexible
 - Using `sprintf` is downright ugly (and inefficient and unsafe)
 - Using non-portable OS code can be especially fun:

```
string s("X = ");  
int dec;  
int sign;  
char* res = _fcvt(x, 3, &dec, &sign);  
if (sign) s += '-';  
s.append(res, dec);  
s += '.';  
s += res + dec;
```

Options

- Change `std::string`
 - This would necessarily mean backward compatibility
 - Which would involve compromises in design and functionality
 - There is strong resistance in the C++ Committee to changing `std::string`
- Add a layer on top of `std::string`
 - For example: `string_ref`
 - This would help a lot in some situations
 - But it would not solve the memory problems
- Write a new string library from scratch
 - This means a fresh start with no compromises due to backward compatibility
 - C++11 should be widely available by the time the library is ready to use
 - I believe the time is right for this to happen

The Solution

- Efficient
 - As with `std::string`, speed is a key consideration
 - Unlike `std::string`, efficiency of memory is also a key consideration
- Powerful
 - Programming should be intuitive and easy
 - Support all common operations in convenient, modern ways
 - Be very flexible without trying to do everything
- Compatible
 - Strings in a program should work together with each other and with other kinds of strings and existing functions
 - Be as similar as possible to `std::string` without introducing compromises
 - Have strong support for Unicode
- Useful
 - Offer an alternative to `string`, `wstring`, and CRT string handling
 - For most programs, all strings should be covered
 - For almost all programs, most strings should be covered

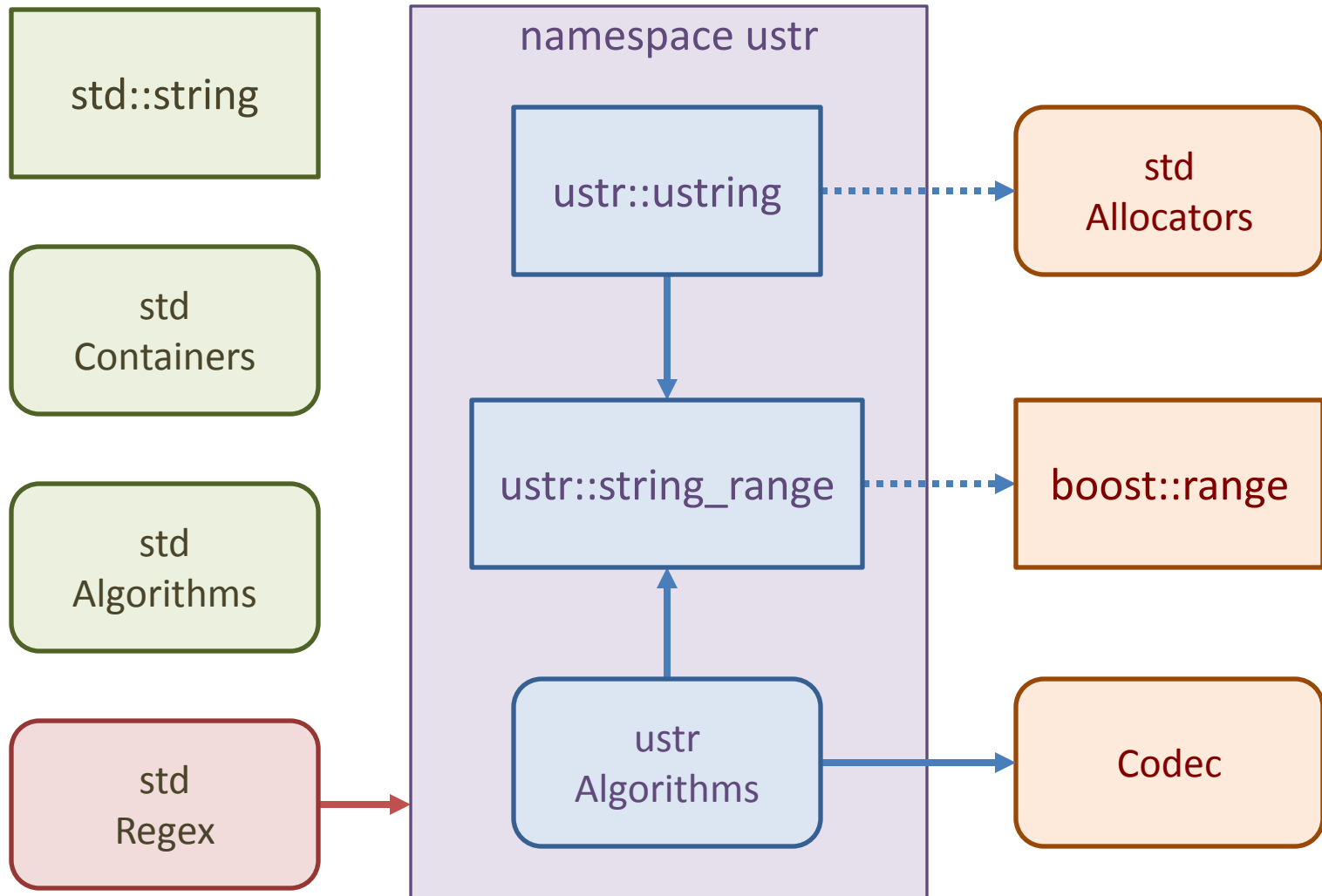
The Solution is Not

- A container—not quite
 - Most container properties are supported
 - It does not quite fully match an STL container concept due to some small differences
- A drop-in replacement for `std::string`
 - Full support of `std::string` functionality would mean supporting more than one metaphor
- A general purpose tool
 - The element type is not a parameter, it is an implementation-defined character type specific to the encoding
 - The size is limited (sort of) by the use of signed int size and position types
- Trying to be all things to all people
 - The goal is to solve a large class of very common string problems
 - Not trying to solve all text-related problems
 - Not a superset of all other string classes; for instance, it is not a rope
 - If you want a repository for your text editor, use a rope
 - If you want to a general container, use `std::string` or some other container

The Objections

- We don't need more stinkin' strings
 - I agree: we need fewer strings
 - To get there we need one that handles more situations more elegantly
- We have SGI Rope
 - Ropes optimize modification of very large strings
 - My concern is memory and speed efficiency for small strings or large, rarely modified strings
- We've been doing OK with `std::string`
 - `std::string` will probably never go away, but I believe `std::string` is no longer sufficient
 - My goal is to make `std::string` obsolescent (new code would be better served by the new string)
- But what about `string_ref`?
 - The ideas behind `string_ref` are incorporated into the `string_range` class
 - `string_ref` is not needed for this new string
 - Would be very nice for maintaining `std::string` code
- Anyway, strings should be immutable
 - Immutability has performance costs and is not compatible with embeddability
- So where's the library?
 - Did you see the first slide?

The Ustring Library



What's in a name?

- **Unicode string**
 - Too limiting
- **Unified String**
 - Too unlimited
- **Ultimate string**
 - Too pretentious
- **Überstring**
 - Too cute
 - Unless you speak German, in which case too pretentious
- **Universal String**
 - Maybe
- **Useful String**
 - That's the idea
- Got a better name?
 - Doesn't have to start with U
 - Let me know (but not now)

The ustring Class Template

- Use a ustring wherever ownership of text is required
- One class template with several parameters
 - You will typically typedef several different variants for your application
- Template Arguments
 - Specify internal representation and encoding
 - Dictate the memory management strategy: local vs. heap (nothing to do with allocators)
 - Allocators may need to be added for Standard compliance to handle heap allocation
- Members
 - All length-modifying operations (e.g. Trim)
 - Some others are included for convenience (e.g. To Upper for simple encodings)

The ustring Class Template

- Internal representation
 - Character type
 - Specified by the Encoding parameter
 - Implementation defined
 - Not user defined because it is a low-level concept and this is a high-level abstraction
 - Character encoding scheme
 - Specified by the Encoding parameter
 - There are 5 choices—any others *may* require using a different tool
 - Text Storage
 - Contiguous
 - Null terminated (embedded nulls are OK except for the zero-overhead version)
 - Size type
 - Both size and difference types are int
 - This is very deliberate: should be the fastest native type
 - If you need more characters, you probably need a different tool anyway

The Template Parameters

```
template<int ENCODING, int FIXED_SIZE, int GROW_TYPE, int GROW_INCREMENT>  
class ustring;
```

- Encoding
 - This parameter dictates both the assumed encoding and the underlying data type
 - ASCII char native 8 bit encoding (e.g. CP-1252)
 - UCS2 wchar_t native 16 bit encoding (e.g. UCS-2)
 - UTF8 unsigned char UTF-8 encoding
 - UTF16 char16_t UTF-16 encoding
 - UTF32 char32_t UTF-32 encoding
- Fixed Size
 - Specifies the size of the local (vs. heap) allocation in elements
 - Includes the null terminator
 - May be zero to indicate heap-only allocation
- Grow Type
 - Controls the management of memory and the size vs. speed tradeoff
 - ZERO_OVERHEAD, FIXED, LINEAR, EXPONENTIAL
- Grow Increment
 - Specifies the amount to grow for Linear and Exponential growth
 - Linear: capacity increases by the Grow Increment in elements
 - Exponential: capacity increases by 1 / Grow Increment

Memory Management

- Zero Overhead
 - Fixed size must be positive
 - Byte compatible with C arrays
 - No cached size
 - Capacity = fixed size
 - `size()`, `end()`, `string_range()` etc. are $O(n)$
 - Cannot have embedded nulls
 - Example: fixed size = 20, size = 19, capacity = 20

| | | | |
|---|---|--------------|--------------|
| W | e | l | c |
| o | m | e | <i>space</i> |
| t | o | <i>space</i> | C |
| + | + | <i>space</i> | N |
| o | w | ! | <i>null</i> |

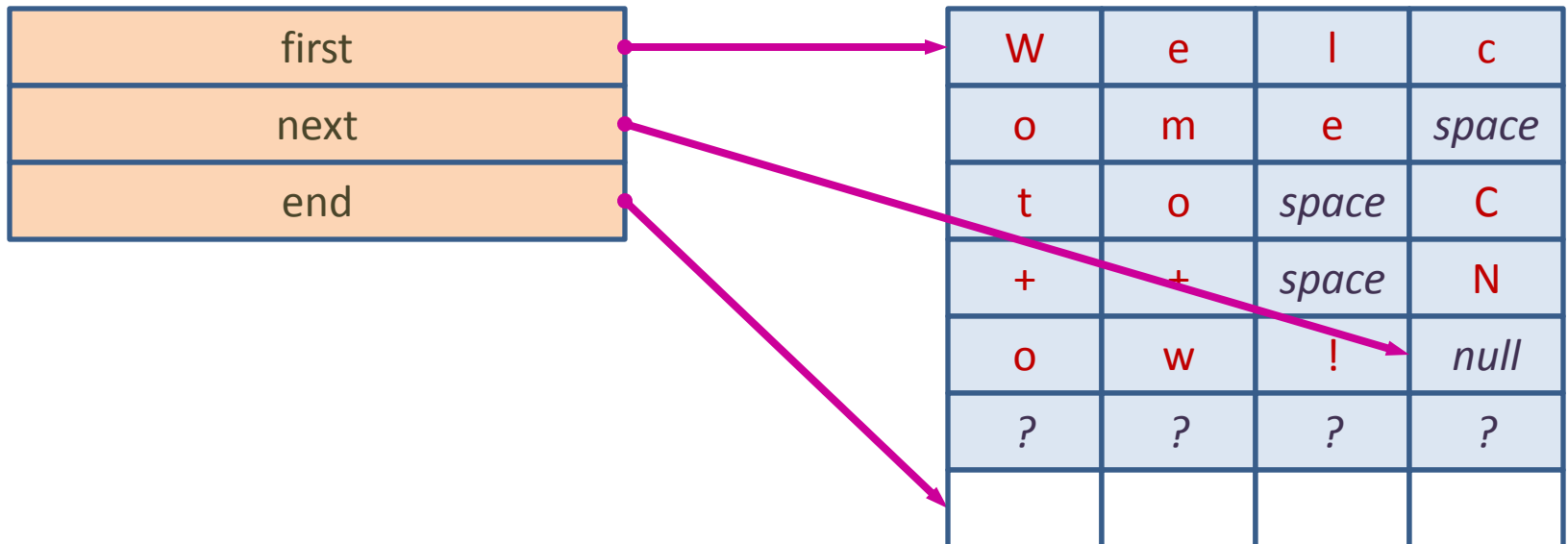
Memory Management

- Fixed
 - Fixed size must be positive
 - Includes a cached size
 - Capacity = fixed size
 - `size()`, `end()`, `string_range()` etc. are $O(1)$
 - Example: fixed size = 20, size = 19, capacity = 20

| size = 19 | | | |
|-----------|---|--------------|--------------|
| W | e | l | c |
| o | m | e | <i>space</i> |
| t | o | <i>space</i> | C |
| + | + | <i>space</i> | N |
| o | w | ! | <i>null</i> |

Memory Management

- Growable, zero fixed size
 - Includes cached size and capacity
 - Text elements are allocated on the heap
 - `size()`, `end()`, `string_range()` etc. are $O(1)$
 - Example: fixed size = 0, size = 19, capacity = 23



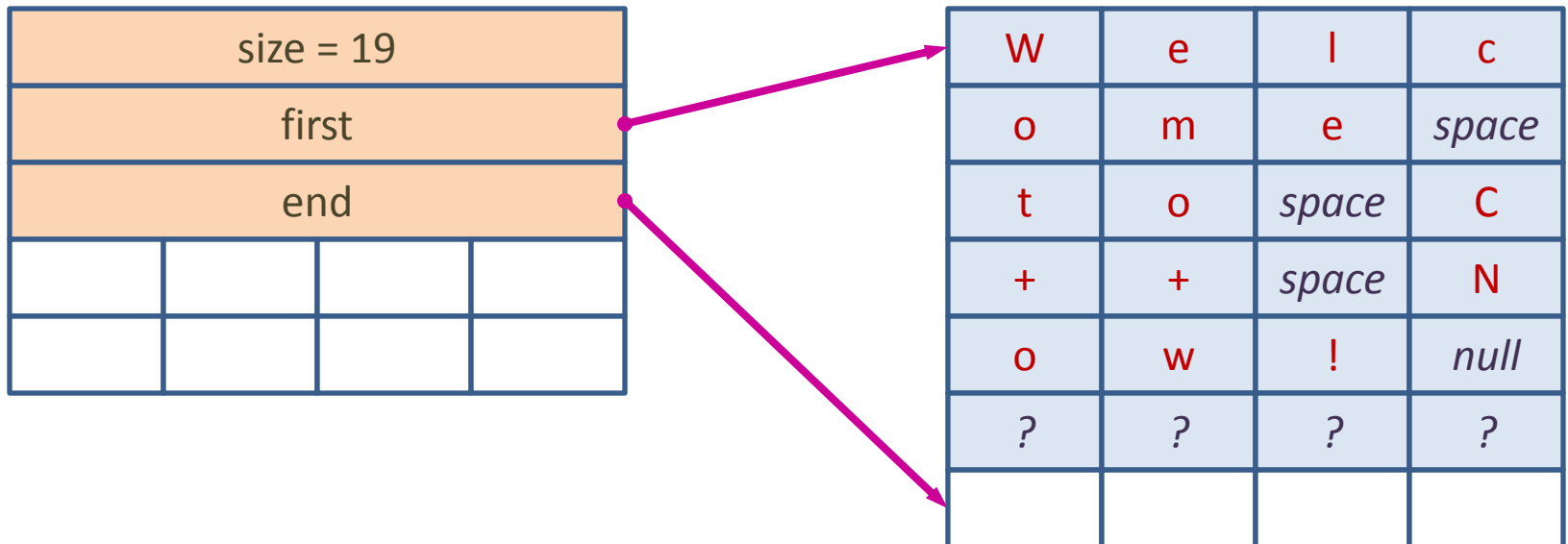
Memory Management

- Growable, positive fixed size
 - Includes cached size and capacity
 - Text elements are allocated locally or on the heap
 - `size()`, `end()`, `string_range()` etc. are $O(1)$
 - Example: fixed size = 16, size = 7, capacity = 15

| size = 7 | | | |
|----------|---|---|--------------|
| W | e | l | c |
| o | m | e | <i>space</i> |
| | | | |
| | | | |

Memory Management

- Growable, positive fixed size
 - Includes cached size and capacity
 - Text elements are allocated locally or on the heap
 - `size()`, `end()`, `string_range()` etc. are $O(1)$
 - Example: fixed size = 16, size = 19, capacity = 23



Basic Members

- Constructors

```
wstring()  
wstring(int count, char_type)  
wstring(const char_type*)  
wstring(string_range)
```

- Assignment

```
wstring& operator=(const char_type*)  
wstring& operator=(string_range)
```

- Conversion

```
operator const char_type*() const  
operator string_range<ENCODING, false>()  
operator string_range<ENCODING, true>()
```

- Free function conversion

```
template<int SZ> wstring& wstring_cast(char (&a)[SZ])
```

Access Members

- Size

| | |
|---|-------------------------------|
| <code>size_type size() const</code> | excluding the null terminator |
| <code>size_type capacity() const</code> | excluding the null terminator |
| <code>bool empty() const</code> | |
| <code>bool heap() const</code> | |

- Iterator access

| | |
|---------------------------------|-----------------|
| <code>char_type* begin()</code> | const/non-const |
| <code>char_type* last()</code> | const/non-const |
| <code>char_type* end()</code> | const/non-const |
| etc... | c versions |

- Element access

| | |
|--|-----------------|
| <code>char_type front()</code> | const/non-const |
| <code>char_type back()</code> | const/non-const |
| <code>char_type operator[](int i)</code> | const/non-const |
| <code>char_type at(int i)</code> | const/non-const |

Insertion Members

- Append

`ustring& operator+=(.)`

`ustring& operator<< (.)`

- Insert

`char_type* insert(int position, char_type)`

`char_type* insert(char_type* where, char_type)`

`char_type* insert(int position, string_range)`

`char_type* insert(char_type* where, string_range)`

- Erase

`void clear()`

`char_type* erase(int position)`

`char_type* erase(char_type* where)`

`char_type* erase(string_range)`

Length Modifying Members

- Editing
 - Trim

```
void trim(char_type char_to_remove)
void trim(const char_type* chars_to_remove)
void trim(string_range chars_to_remove)
```
 - Trim Front, Trim Back
 - Remove
 - Same as Trim but throughout the string
 - Replace
- Formatting
 - Format
 - printf vs. Python-like
 - Implemented as a variadic template rather than a variadic function
 - Field
 - Expand to given length
 - Text is positioned left, center, or right
- Encoding Conversions

Other Members

- Search
 - Find First, Find Last
- Stream
 - Operator <<
 - Operator >>

Ustring Examples

```
ustring<ASCII> us;  
us << "Pi = " << precision(3.14159, 3) << " and UQ = " << 42;  
cout << us;
```

Pi = 3.142 and UQ = 42

| | |
|---|--|
| <pre>char buff[64] = "Welcome"; auto& usc = ustring_cast(buff); usc += '!'; usc.insert(usc.last(), " to C++ Now"); to_upper(substr(usc, 0, 7));</pre> | <p>Welcome</p> <p>Welcome!</p> <p>Welcome to C++ Now!</p> <p>WELCOME to C++ Now!</p> |
|---|--|

String Range

- Overview

- The link between the ustring class and the string algorithms
- Provides interoperability with other kinds of strings
- Is often all you need

- Template Arguments

```
template<int ENC, bool CONST = false> struct string_range;  
template<int ENC> using const_string_range = string_range<ENC, true>;
```

- Encoding

- The same meaning as for ustring

- Const

- True if this range refers to const data

String Range Members

- Constructors

```
string_range()  
string_range(char_type*)  
template<size_t SZ> string_range(char_type (&ar)[SZ])  
string_range(const std::basic_string&)
```

- Content

```
operator bool()  
bool empty()  
size_type size()
```

- Iterator access

```
char_type* operator*()  
char_type* begin()  
char_type* last()  
char_type* end()
```

- Element access

| | |
|-----------------------------|-----------------|
| char_type front() | const/non-const |
| char_type back() | const/non-const |
| char_type operator[](int i) | const/non-const |
| char_type at(int i) | const/non-const |

String Range Members

- Shrink operations

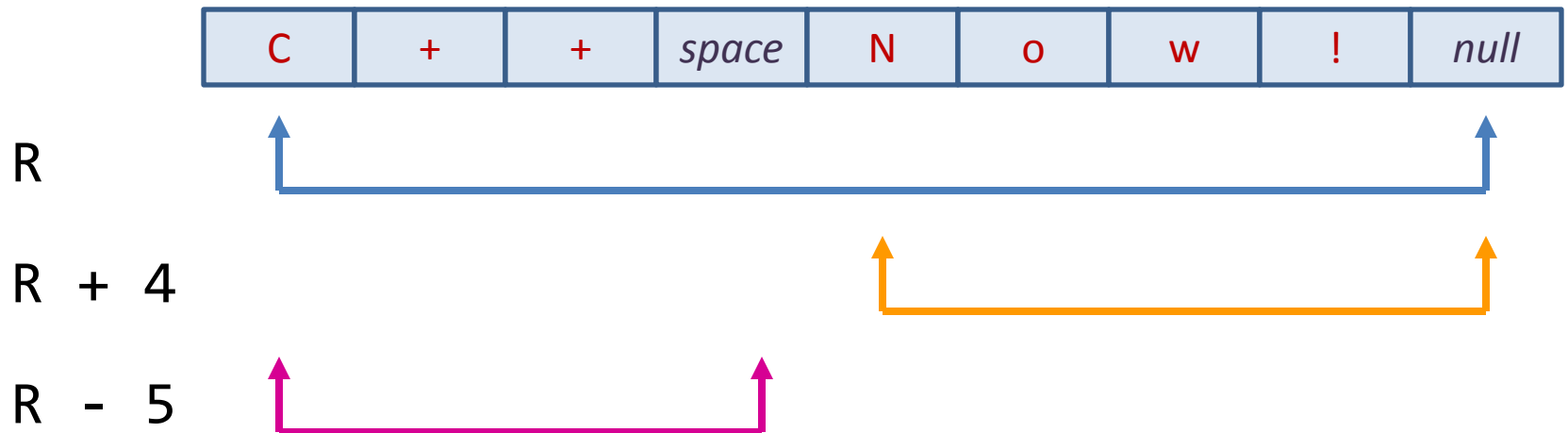
`string_range& operator++()` (not safe)

`string_range& operator++(int)` (not safe)

`string_range& operator+=(int distance)`

`string_range operator+(string_range, int distance)`

etc...

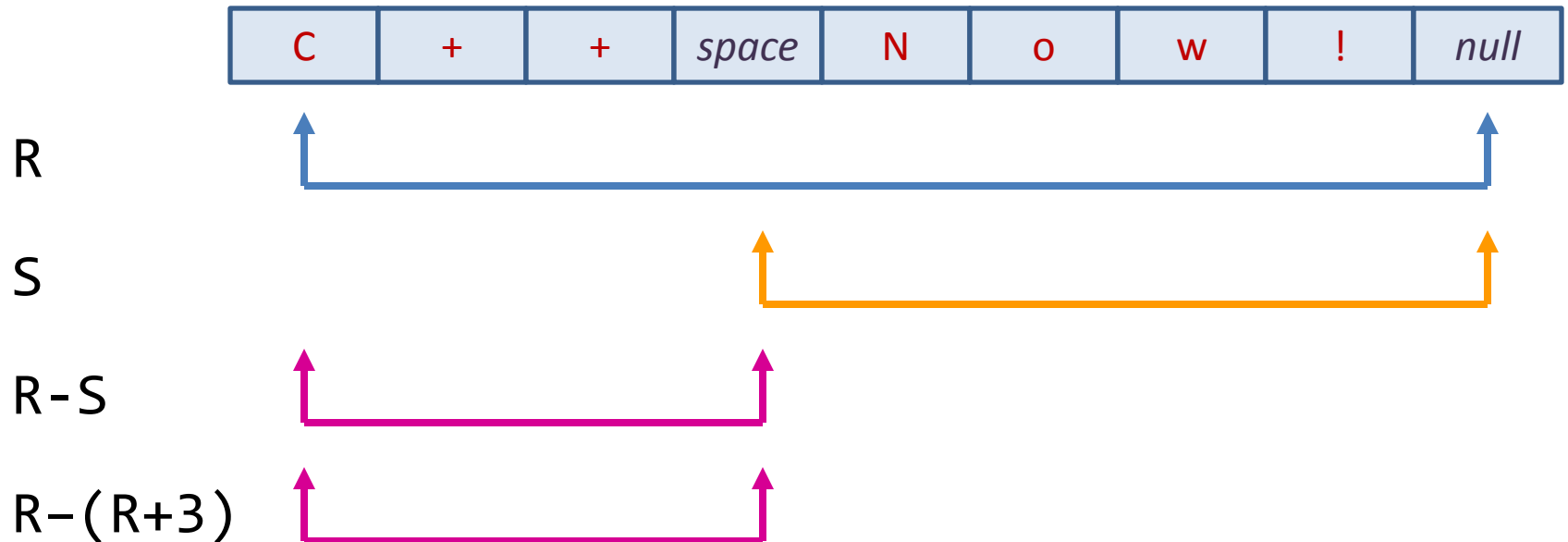


String Range Members

- Shrink operations

`string_range& operator-=(string_range)`

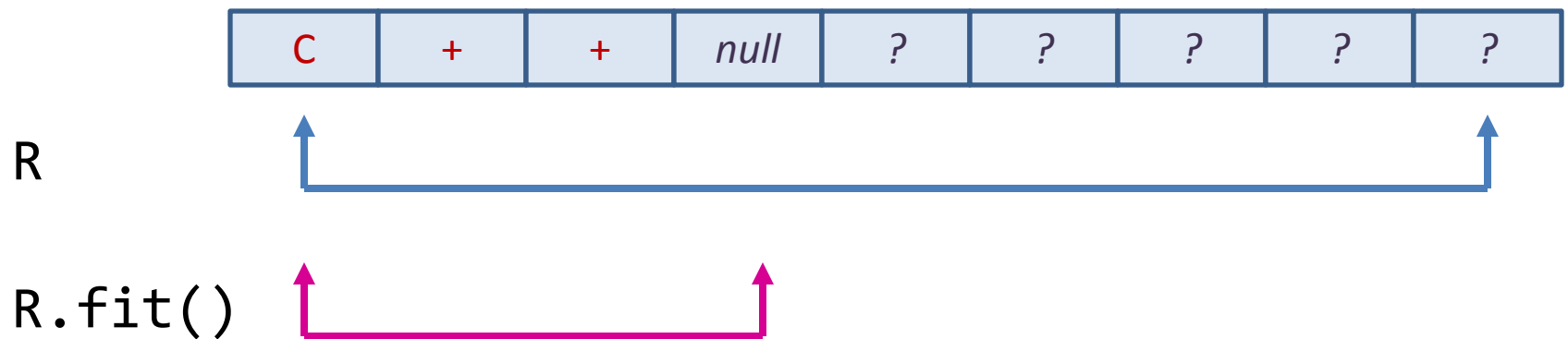
`string_range operator-(string_range, string_range)`



String Range Members

- Shrink operations

`string_range& fit()`



string_range Example

```
string_range<ASCII, true> r = "Welcome to C++ Now!";
```

```
for (; r; ++r)  
    cout << *r;
```

Welcome to C++ Now!

Algorithms

- Goals
 - Composability
 - Compatibility
 - Convenience
 - Performance
- Design
 - Conceptually take a `string_range` and (usually) other arguments
 - Actually use TMP to generate a `string_range` from many types
 - Return
 - `string_range`
 - `bool`
 - May modify the target range

Non-modifying Algorithms

- All
 - Could also be called Make Range
- Substring
 - substr From start to start + length
 - substrp From start to stop
- Trim
 - Takes either `char_type` or a `string_range` to match
 - Defaults to white space
 - Trim Front
 - Trim Back
- Token
 - Takes a `string_range&` and advances it as each token is found
 - Returns a `string_range` that defines the token
 - Takes either `char_type` or a `string_range` to match
 - Non-destructive
- Divide
 - Same as Token, but includes the delimiter in the returned token
 - Nice for breaking text into lines

substr

```
template<typename T>
typename string_range_traits
<typename std::remove_reference<T>::type>::type
substr(T&& t, int start, int length = std::numeric_limits<int>::max())
{
    typename string_range_traits
    <typename std::remove_reference<T>::type>::type
    str(std::forward<T>(t));

    str += start;
    str -= str.size() - length;

    return str;
}
```

substrp

```
template<typename T>
typename string_range_traits
<typename std::remove_reference<T>::type>::type
substr(T&& t, int start, int stop = std::numeric_limits<int>::max())
{
    typename string_range_traits
    <typename std::remove_reference<T>::type>::type
    str(std::forward<T>(t));

    str -= str.size() - stop;
    str += start;

    return str;
}
```

substr Calls

```
cout << substr("Maroon Bells", 7, 5) << endl;
```

```
char buff[64] = "Maroon Bells";  
cout << substr(buff, 7, 5) << endl;
```

```
auto c = "Maroon Bells";  
cout << substr(c, 7, 5) << endl;
```

```
const string s("Maroon Bells");  
cout << substr(s, 7, 5) << endl;
```

```
wstring ws(L"Maroon Bells");  
wcout << substr(ws, 7, 5) << endl;
```

```
string_range<ASCII, true> sr = "Maroon Bells";  
cout << substr(sr, 7, 5) << endl
```

```
ustring<ASCII> us("Maroon Bells");  
cout << substr(us, 7, 5) << endl;
```

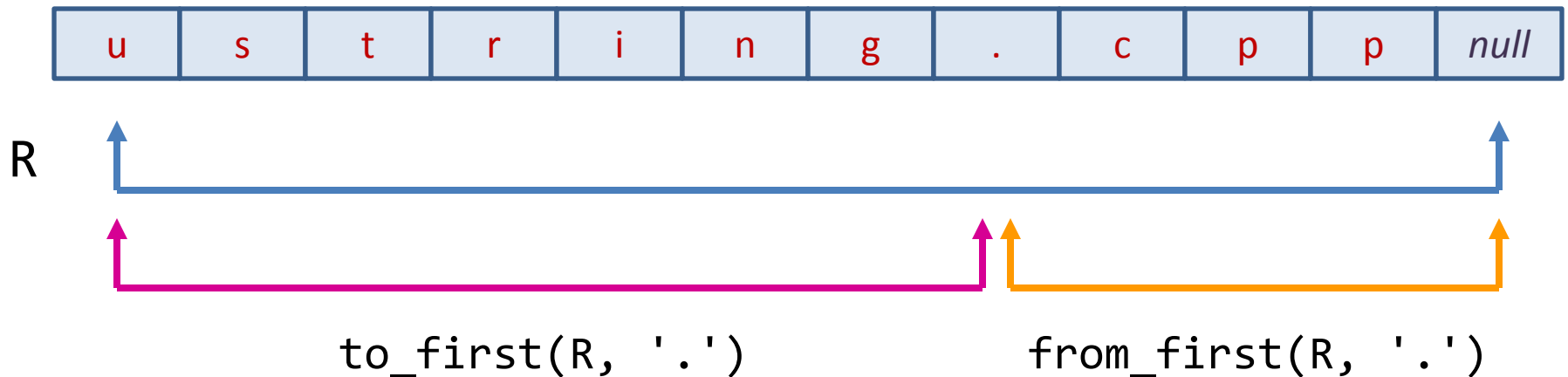
token Call

```
auto data = all("Welcome to C++ Now! in Aspen, CO");  
while (data)  
    cout << token(data, ' ') << endl;
```

Welcome
to
C++
Now!
in
Aspen,
CO

Non-modifying Algorithms

- To First, From First
 - Takes either `char_type` or a `string_range` to search for
 - To in the sense of "up to", from in the sense of "starting from"
 - so `to_first(R, X)` plus `from_first(R, X)` equals `R`
- To Last, From Last
- To First Not, From First Not
- To Last Not, From Last Not



Modifying Algorithms

- Copy
 - Takes two string_ranges, source and destination
 - Only works within one encoding
 - Safe copy
- Replace
 - Takes two characters, replaces all occurrences of one with the other
 - Only works within one encoding
 - For cross-encoding replace, the ustring member is required because of length changes
- Reverse
 - Reverses the range in place
- To Upper / To Lower
 - Makes changes in place
 - Only works within one encoding
 - For cross-encoding replace, the ustring member is required because of length changes

Comparison Algorithms

- Equal, Equal NC
 - Binary equality
 - NC uses simple (fast) conversion
- Less, Less NC, Greater, Greater NC
 - Binary comparison
- Compare, Compare NC
 - Binary comparison
 - Returns -1, 0, 1
- Unicode and Locale
 - Support for more intelligent comparison

Extract File Title

```
auto path = all("D:\\Code\\Ustring\\Source\\Heaponly.h");  
cout << (from_last(path - from_last(path, '.'), '\\') += 1);
```

Heaponly

```
cout << to_last(from_last(path, '\\') += 1, '.');
```

Heaponly

Algorithm Composition

```
cout <<
  to_first(
    trim_front(
      from_first(
        from_first("Colorado rocky mountain high", 'k'), ' '
      )
    ), ' '
  );
```

mountain

Switch on File Type (MFC)

```
CString ext(lpszPathName);
int ext_len = ext.ReverseFind('.');
ext = ext.Right(ext_len == -1 ? 0 : ext.GetLength() - (ext_len + 1));

if (ext.CompareNoCase("top") == 0)          // If this is a TOP file.
{
}
else if (ext.CompareNoCase("rr") == 0)      // If this is an RR file.
{
}
```

Switch on File Type (STL)

```
string ext(lpszPathName);  
ext = ext.substr(ext.find_last_of('.') + 1, ext.npos);  
  
if (_stricmp(ext.c_str(), "top") == 0)    // If this is a TOP file.  
{  
}  
else if (_stricmp(ext.c_str(), "rr") == 0) // If this is an RR file.  
{  
}
```

Switch on File Type (ustring)

```
auto ext = ustr::from_last(lpszPathName, '.') += 1;

if (equal_nc(ext, "top"))          // If this is a TOP file.
{
}
else if (equal_nc(ext, "rr"))      // If this is an RR file.
{
}
```

Interoperability with std::string

```
std::string boostcon("BoostCon");  
wstring<ASCII> cppnow("C++ Now!");  
  
cout << greater(cppnow, boostcon);
```

true

The Discussion

- Ground rules
 - We have 45 minutes and n people, so each person gets $t = 45/n$
 - Once everyone has been heard, people can have second turns
 - Show of hands who might like to participate so we can calculate t
 - The goal is to get lots of good ideas out on the table
 - We do not need to solve every problem
 - We do not need to convince anyone of anything
 - We do not need to reach consensus
- Focus
 - Big picture
 - Architecture
 - API design
 - Use cases
- Examples
 - I can show some real code examples as we go

Acknowledgements and Thanks

- Beman Dawes
 - Reviewed the library design in depth and made many helpful suggestions about the library and this presentation
- Jeffrey Yasskin
 - Presented proposals to the C++ Committee on `string_ref` and ranges, discussed this library with me, and helped to convince me of the value of the range-based design
- David Abrahams
 - Discussed the idea with me and encouraged me to give a presentation at this early stage in the library's development
- You
 - For all the great ideas you are about to contribute