Document Number: X3J16/92-0006
WG21/N0084
Date: February 4, 1992
Project: Programming Language C++
Reply To: Uwe Steinmueller
stein%ap542@ztivax.siemens.com

A String Class in C++

Uwe Steinmueller

SNI AP 44

Siemens Nixdorf Informationssysteme AG 1992

1. Purpose of this paper

This paper is not intended to be a final draft but should be the bases of the discussion on the string class functionality in the library WG.

2. A String Class for C++ has to deal with the following problems:

- Value or pointer semantics for strings. This version assumes value semantics. (Of course an implemenation may choose using reference counted pointers and implementing the value semantics with a copy-on-write strategy).
- Temporaries (lifetime and performance overhead).
- Conversion to char* (dangling pointers and breaking class boundaries). As we believe there is no realistic way to leave it out (especially interfacing with C) we prevent the implicit casting to char*.
- National language support (this version only for single byte character sets).
- Benefit and cost of operator overloading (readability and understandability of code)
- Storing of ASCII 0x00 an arbitrary positions in a string. We decided to store 0x00 in a string as it is not seldom needed and otherwise would limit the string class unnecessarily.
- Practical tests have shown that strings, which allocate memory in some sort of chunks and therefore need an extra value to store the current capacity, can perform significantly better in some sort of programs. We do not want to force an implementation to use this stategy, but as the main reason for a standard is to provide a better portability, our approach tries to support both ways. An implementation is allowed to ignore all capacity requests. Capacity is just a hint to the implemention it might use or not. By this an application is portable across different implementations. A default parameter for StringCapacity is added to most constructors as the exact or approximate future string size is often known at construction time.
- Search using regular expressions should be handled in an extra class.
- A class Stringstream should be provided (not specified here).
- Some might argue that the presented class here is far too much featurism and many of the extra functions can easily be added by the implementors or the users. But exactly this is the problem: if we know that something will be a "common" situation in practice, the standard should guarantee that there is an acceptable (what ever this means) solution with the

standard.

We agree that a standard with many member functions is more difficult to understand than a more minimal solution, but we believe that it is far more difficult to understand all the different extensions to the standard.

3. Public interface of the string class

The class Substring is introduced because it is common practice and of notational convenience.

The type SLT is used for the string length as we need an extra value to indicate an invalid position (NPOS). This is needed especially for return values of the find operations. A posible candidate is "unsigned int" and defining NPOS = UINT_MAX, then a check for a valid position is just one compare (pos < NPOS).

Used names for variables:

String:	s, s1, s2
Pointer to C string:	cs
Pointer to char* buffer:	cb
Substring:	sub
SLT:	pos, rep, n
Char:	c
Ostream:	os
Istream:	is
StringCapacity:	ic

```
 //
 // some needed types
 //

typedef <some integral type> SLT;  // for String position values and length
extern  const SLT NPOS;            // some value indicating an invalid SLT
typedef int bool;

class String; class Substring;

 //
 // Class StringCapacity
 // class StringCapacity {
    //
    // friends
    //
    friend class String;

public:
    StringCapacity(SLT n);
    ~StringCapacity(); };

 //
 // public interface of String class
 //

class String {
    //
    // Exceptions: OutOfMemory, OutOfRange
    //
    //
    // friends
    //
    friend class Substring;

public:
    //
    // constructors
    //

    String();
    String(const String& s, StringCapacity ic = 0);
    String(const char* cs);
    String(const char* cb, SLT n, StringCapacity ic = 0);
    String(char c, SLT rep = 1, StringCapacity ic = 0);
    String(const Substring& sub);

    String(StringCapacity ic);

    //
    // destructor
    //

    ~String();

    //
    // Assignemt (value semantics)
    //

    String& operator=(const String& s);
    String& operator=(const char* cs);
    String& assign(char c, SLT rep = 1);
    String& assign(const char* cb, SLT n);
    String& operator=(const Substring& sub);
    String& operator=(char c);                  // for convenience


    //
    // Concatenation
    //

    String& operator+=(const String& s);
    String& operator+=(const char* cs);
    String& append(const char* cb, SLT n);
    String& append(char c, SLT rep = 1);
    String& operator+=(const Substring& sub);
    String& operator+=(char c);                 // for convenience

    friend String operator+(const String& s1, const String& s2);
    friend String operator+(const char* cs, const String& s);
    friend String operator+(const String& s, const char* cs);
    friend String operator+(char c, const String& s);
    friend String operator+(const String& s, char c);

    //
    // Predicates
    //

    friend bool operator==(const String& s1, const String& s2);
    friend bool operator!=(const String& s1, const String& s2);
```

```
//
// Comparison
//

friend int compare(const String& s1, const String& s2);
friend bool operator>(const String& s1, const String& s2);
friend bool operator>=(const String& s1, const String& s2);
friend bool operator<(const String& s1, const String& s2);
friend bool operator<=(const String& s1, const String& s2);

//
// Comparison   (depend on collating sequences)
//

int strcoll(const String& s) const;
String& strxfrm();

//
// Insertion at some pos
//

String& insert(SLT pos, const String& s);
String& insert(SLT pos, const char* cs);
String& insert(SLT pos, const char* cb, SLT n);
String& insert(SLT pos, char c, SLT rep);

//
// Removal
//

String& remove(SLT pos, SLT n);

//
// Replacement at some pos
//

String& replace(SLT pos, SLT n, const String& s);
String& replace(SLT pos, SLT n, const char* cs);
String& replace(SLT pos, SLT n, const char* cb, SLT n);
String& replace(SLT pos, SLT n, char c, SLT rep);

//
// Subscripting
//

char& operator[](SLT pos);
const char operator[](SLT pos) const;

//
// Search
//

SLT find(char c, SLT pos = 0) const;
SLT find(const String& s, SLT pos = 0) const;
SLT find(const char* cs, SLT pos = 0) const;
SLT findReverse(char c, SLT pos = NPOS) const;
SLT findReverse(const String& s, SLT pos = NPOS) const;
SLT findReverse(const char* cs, SLT pos = NPOS) const;

//
```

```
    // Substrings
    //

    Substring operator() (SLT pos, SLT n);
    String getSubstring(SLT pos, SLT n) const;

    //
    // I/O
    //

    friend ostream& operator<<(ostream& os, const String& s);
    friend istream& operator>>(istream& is, String& s);
    friend istream& getline(istream& is, String& s, char c = '\n');

    //
    // Miscellanious
    //

    // length

    SLT length() const;

    // copy to C buffer

    SLT copy2c(char* cb, SLT n, SLT pos = 0) const;

    // get pointer to internal character array

    const char* getCPtr() const;

    // upper/lower

    String& toUpper();
    String& toLower();

    // Capacity

    SLT capacity();
    SLT capacity(StringCapacity ic);
 };

class Substring {
    friend class String; public:

    //
    // Assignment
    //

    void operator=(const String&);
    void operator=(const char*);
    void assign(const char*, SLT n);
    void assign(char c, SLT rep);
 };
```

4. Description of the public String member functions

4.1. Constructors

Declaration:

`String()`

Synopsis:

Default constructor creates String of length zero.

Pre-conditions:

`None`

Post-conditions:

`length() == 0`

Result:

None

Exceptions:

None (OutOfMemory depending on implementation)

Declaration:

`String(const String& s, StringCapacity ic = 0)`

Synopsis:

Copy constructor creates a String with the value copy of the String s.

The value of ic can be used by the implementation. If ic < length() the internal capacity will be >= length().

Pre-conditions:

`None`

Post-conditions:

```
length() == s.length()
memcmp(getCPtr(), s.getCPtr(), s.length()) == 0
```

Result:

None

Exceptions:

OutOfMemory

Declaration:

`String(const char *cs)`

Synopsis:

Creates a String containing the characters of the C-string. If cs is 0 an empty String is created.

No default value for the capacity as this can be done using the constructor String(char* cb, SLT n, StringCapacity ic = x). (String(cs, strlen(cs), x))

Pre-conditions:

`None`

Post-conditions:

```
if(cs != 0)
    length() == strlen(cs)
else
    length() == 0
memcmp(getCPtr(), cs, length()) == 0
```

Result:

None

Exceptions:

OutOfMemory

Declaration:

```
String(const char *cb, SLT n, StringCapacity ic = 0)
```

Synopsis:

Creates a String containing the first n elements of the buffer pointed to by cb.

The value of ic can be used by the implementation. If ic < n the internal capacity will be >= n.

Pre-conditions:

```
n != NPOS
```

Post-conditions:

```
length() == n
memcmp(getCPtr(), cb, n) == 0
```

Result:

None

Exceptions:

OutOfMemory, OutOfRange

Declaration:

```
String(char c, SLT rep = 1, StringCapacity ic = 0)
```

Synopsis:

Creates a String containing rep times character c.

The value of ic can be used by the implementation. If ic < rep the internal capacity will be >= rep.

Pre-conditions:

```
rep != NPOS
```

Post-conditions:

```
length() == rep
for(i = 0; i < rep; i++)
    *(getCPtr() + i) == c
```

Result:

None

Exceptions:

OutOfMemory, OutOfRange

Declaration:

`String(const Substring& sub)`

Synopsis:

Creates a String containing a content copy of sub.

No capacity value as default because this constructor most of the time is used for implicit casts form Substring to String

Pre-conditions:

`None`

Post-conditions:

```
length() == Length(sub)
memcmp(getCptr(), subs internal start, length()) == 0
```

Result:

None

Exceptions:

None

Declaration:

`String(StringCapacity ic)`

Synopsis:

Constructor creates String of length zero. The implementation can use the capacity value as a hint for use of internal storage. If ic < length() the value ic is ignored.

Pre-conditions:

`ic != NPOS`

Post-conditions:

`length() == 0`

Result:

None

Exceptions:

OutOfMemory, OutOfRange

4.2. Destructor

Declaration:

`~String();`

Synopsis:

Destructing the String and freeing all unneeded memory.

Pre-conditions:

```
None
```

Post-conditions:

```
None
```

Result:

None

Exceptions:

None

4.3. Assignment

Declarations:

```
String& operator=(const String& s)

analogous:

String& operator=(const char *cs)
String& assign(char c, SLT rep = 1)
String& assign(const char *cb, SLT n)
String& operator=(const Substring& sub)
String& operator=(char c)
```

Synopsis:

Frees old content and creates a copy of s if &s != this. Returns a reference to the target String.

Pre-conditions:

```
None
```

Post-conditions:

```
length() == s.length()
memcmp(getCPtr(), s.getCPtr(), length()) == 0
```

Result:

Reference to String

Exceptions:

OutOfMemory

4.4. Concatenation

Declarations:

```
String& operator+=(const String& s)

analogous:

String& operator+=(const char *cs)
String& append(const char *cb, SLT n)
String& append(char c, SLT rep = 1)
```

```
String& operator+=(const Substring& sub)
String& operator+=(char c)
```

Synopsis:

Append content of String to the target String and return a reference to the target.

Pre-conditions:

```
n != NPOS, rep != NPOS
```

Post-conditions:

```
length() == s.length() + (oldlength = Length(target on entry))
memcmp(getCPtr() + oldlength, s.getCPtr(), s.length()) == 0
```

Result:

Reference to String

Exceptions:

OutOfMemory, (OutOfRange if SLT parameter is used)

Declarations:

```
friend String operator+(const String& s1, const String& s2)
```

```
analogous:
```

```
friend String operator+(const char* cs, const String& s)
friend String operator+(const String& s, const char* cs)
friend String operator+(char c, const String& s)
friend String operator+(const String& s, char c)
```

Synopsis:

Concatenate Strings with Strings, C-strings and characters. The many overloaded functions reduce the necessity for creating temporary objects. For these functions it is essential that the lifetime of temporaries is at minimum the lifetime of the statement.

These functions are not needed for there functionality. They are current practice and nice for their convenience. But the users and implementors should be aware of their problems.

There might be in some cases a non acceptable performance overhead due to creation of temporaries. Speciall care must be taken with the use of the getCPtr() member functions.

```
char *p = ("/foo" + '/' + "foo.c").getCPtr();
open(p);         // p is not guarantied to be valid
```

Returns a String holding the result.

Pre-condition

```
None
```

Post-conditions:

```
String s = s1 + s2;
s.length() == (s1.length() + s2.length())
memcmp(s.getCPtr(), s1.getCPtr(), s1.length()) == 0
memcmp(s.getCPtr() + s1.length(), s2.getCPtr(), s2.length()) == 0
```

Result:

String

Exceptions:

OutOfMemory

4.5. Predicates

Declarations:

```
friend bool operator== (const String& s1, const String& s2)
```

```
anlogous:
```

```
friend bool operator!= (const String& s1, const String& s2)
```

Synopsis:

Test for equality (not equality) of String s1 with the String s2. Returns a boolean value.

Pre-conditions:

None

Post-conditions:

None

Result:

Bool

Exceptions:

None

4.6. Comparison

Declaration:

```
friend int compare(const String& s1, const String& s2)
```

Synopsis:

Compares String s1 with the String s2. The implementation should use the C function memcmp() to get the same results as with C. It returns an integer less than, equal to, or greater than 0, according to s1 is lexicographically less than, equal to, or greater than s2. This function is a friend function as there are many situations where one would like to pass a compare function to to some other function (e.g. to a sort function of a container class).

Pre-conditions:

```
None
```

Post-conditions:

```
None
```

Result:

int

Exceptions:

None

Declarations:

```
friend bool operator> (const String& s1, const String& s2)
```

analogous:

```
friend bool operator>= (const String& s1, const String& s2)
f   nd bool operator< (const String& s1, const String& s2)
f   nd bool operator<= (const String& s1, const String& s2)
```

Synopsis:

Returns the result of the lexicographically compare of the Strings s1 and s2. ng pare() s.o.)

Pre-conditions:

None

Post-conditions:

None

Result:

Bool

Exceptions:

None

4.7. Comparison depending on the status of locale category LC LATE

D ation:

```
in  rcoll(const String& s)
```

Synopsis:

Returns the result of strcoll(this->getCPtr(), s.getCPtr()). Both Strings are treated like null terminated C-strings. The implementation has to guarantee that both strings can be handled this way (see getCPtr()). For a more detailed description look to the ANSI C Standard. As there may be some transformations necessary an OutOfMemory exception might be thrown.

Pre-conditions:

There is a LC_COLLATE local selected. (see ANSI C)

Post-conditions:

```
strcoll(s) == strcoll(getCPtr(), s.getCPtr())
```

Result:

Bool

Exceptions:

OutOfMemory

Declaration:

```
String strxfrm()
```

Synopsis:

Returns a String which is the transformed version of the target String that the following is guaranteed:

```
    String s1, s2;
    .. s1, s2 get some content
    String ts1 = s1.strxfrm();
    String ts2 = s2.strxfrm();

    assert(s1.strcoll(s2) == strcmp(ts1.getCPtr(), ts2.getCPtr());
    assert(s1.strcoll(s2) == strcoll(s1.getCPtr(), s2.getCPtr());
```

Pre-conditions:

`There is a LC_COLLATE local selected. (see ANSI C)`

Post-conditions:

`None`

Result:

String

Exceptions:

OutOfMemory

4.8. Insert operations

Declarations:

`String& insert(SLT pos, const String& s)`

`analogous:`

```
String& insert(SLT pos, const char* cs)
String& insert(SLT pos, const char* cb, SLT n)
String& insert(SLT pos, char c, SLT rep)
```

Synopsis:

Insert the String s at Position pos into the target String. If pos >= s.length() an OutOfRange exception is thrown. A Reference to the modified target String is returned.

Pre-conditions:

`pos < (oldlength = length())`

Post-conditions:

```
length() == s.length() + oldlength
memcmp(getCPtr() + pos, s.getCPtr(), s.length()) == 0
```

Result:

Reference to String

Exceptions:

OutOfMemory, OutOfRange

4.9. Removal

Declaration:

```
String& remove(SLT pos, SLT n)
```

Synopsis:

>From the target String len characters starting at position pos are removed. If n == NPOS then len = length() - pos else len = min(n, length() - n). A reference to the result is returned.

Pre-conditions:

```
pos != NPOS; pos < (oldlength = length())
```

Post-conditions:

```
length() == length() - len
```

Result:

Reference to String

Exceptions:

OutOfMemory, OutOfRange

4.10. Replace operations

Declarations:

```
String& replace(SLT pos, SLT n, const String& s)
```

```
analogous:

String& replace(SLT pos, SLT n, const char* cs)
String& replace(SLT pos, SLT n, const char* cb, SLT n)
String& replace(SLT pos, SLT n, char c, SLT rep)
```

Synopsis:

s.replace(pos, n, s) is exactly the same as s.remove(pos, n) followed by s.insert(pos, s) but it can be implemented more efficiently and is more conveniently. That *this(pos, n) = s (using Substrings) does exactly the same. This version may imply lesser overhead and some users might prefere this style.

Pre-conditions:

```
pos < (oldlength = length())
```

Post-conditions:

```
s1 = s;
s1.remove(pos, n);
s1.insert(pos, s); s1 == s.replace(pos, n, s)
```

Result:

Reference to String

Exceptions:

OutOfMemory, OutOfRange

4.11. Subscripting

Declarations:

```
char& operator[](SLT pos)
const char operator[](SLT pos) const
```

Synopsis:

Returns a reference to the character at position pos. The second form is used for const Strings. If pos is invalid an OutOfRange Error is thrown.

Pre-conditions:

```
pos < length()
```

Post-conditions:

```
None
```

Result:

Reference to char (or char, for the const version)

Exceptions:

OutOfRange, (implementation dependant OutOfMemory)

4.12. Find operations

Declarations:

```
SLT find(char c, SLT pos = 0) const
```

```
analogous:

SLT find(const String& s, SLT pos = 0) const
SLT find(const char* cs, SLT pos = 0) const
SLT findReverse(char c, SLT pos = NPOS) const
SLT findReverse(const String& s, SLT pos = NPOS) const
SLT findReverse(const char* cs, SLT pos = NPOS) const
```

Synopsis:

All find member functions search for a String, character or C string in the target String. If pos is a valid index and the searched object is found, the position is returned or otherwise the value NPOS.

FindReverse searches backwards, NPOS indicates a start at the end of the String.

Pre-conditions:

```
pos != NPOS (for all find versions)
```

Post-conditions:

```
None
```

Result:

SLT

Exceptions:

None (OutOfMemory depending on the search algorithms used)

4.13. Substrings

Declarations:

```
Substring operator() (SLT pos, SLT n)
String getSubstring(SLT pos, SLT n) const
```

Synopsis:

Operator() creates a Substring or getSubstring creates a String with the content of the characters in the target String ranging from pos to pos + len. If n == NPOS then len = length() - pos else len = min(n, length() - pos. The member function getSubstring can be used for const Strings.

An OutOfRange exception is thrown if pos == NPOS or pos >= length().

Pre-conditions:

```
pos < length(); n != NPOS;
```

Post-conditions:

```
None
```

Result:

Substring (String)

Exceptions:

OutOfRange, OutOfMemory

4.14. String input/output operations

Declarations:

```
friend ostream& operator<<(ostream& os, const String& s)
friend istream& operator>>(istream& is, String& s)
friend istream& getline(istream& is, String& s, char c = '0)
```

Synopsis:

Operator<< outputs to the ostream os all characters of String s. Also characters containing 0x00 will be written to os.

Operator>> inserts all characters up to the next white space, EOF or error not including the white space to the String s.

Getline creates a String s containing all character up to the next character c, EOF or error (not containing c itself).

Pre-conditions:

```
A valid stream
```

Post-conditions:

```
None
```

Result:

ostream& (istream&)

Exceptions:

(see exceptions of the iostream library)

4.15. Miscellanious

Declaration:

`SLT length() const`

Synopsis:

Returns the length of the String. As characters 0x00 can be stored in a String length() might be != strlen(getCPtr()).

Pre-conditions:

`None`

Post-conditions:

`None`

Result:

SLT

Exceptions:

None

Declaration:

`SLT copy2c(char* pc, SLT n, SLT pos = 0) const`

Synopsis:

If pc is zero the value of length() - pos + 1 is returned. Otherwise len = min(n - 1, length() - pos) characters starting at pos are copied to the area pointed to by pc. *(pc + len) is set to zero to guarantee that the string is always null-terminated. The value of len + 1 is returned.

If pos is out of range or n == NPOS the OutOfRange exception is thrown.

Pre-conditions:

`pos < length(); //the area pointed to by pc must hold n characters`

Post-conditions:

`*(pc + len) == 0`

Result:

SLT

Exceptions:

OutOfRange

Declaration:

`const char* getCPtr() const`

Synopsis:

Returns a char* pointer to the internal representation of the String. Nearly all non const member functions may invalidate this pointer. Do not use this function on temporaries. This function guarantees that the string is null-terminated. If the implementation uses a copy-on-write mechanism there should be a memberfunction being provided to get a unique copy of the string. A cast to char* (casting constness away) should not be used, the subscripting functions is to be prefered.

Pre-conditions:

`The String is not a temporary`

Post-conditions:

`pc points to a null-terminated string`

Result:

char *

Exceptions:

None

Declarations:

```
String& toUpper()
String& toLower()
```

Synopsis:

Converts the String to upper or lower case. The implementation should use the ANSI C functions toupper() or tolower() to be conform with the C locales. The functions return references to their target Strings.

Pre-conditions:

`None`

Post-conditions:

`None`

Result:

Reference to String

Exceptions:

None

Declaration:

```
SLT capacity()
SLT capacity(StringCapacity ic)
```

Synopsis:

Capacity() returns a value which is decided by the implementation to indicate the current internal storage size. The returned value is always greater or equal then lenght(). The second function gives a hint to the implementation and returns the new capacity. A value ic < length() is ignored.

Pre-conditions:

`ic != NPOS`

Post-conditions:

`return value >= length() and String content unchanged`

Result:

SLT

Exceptions:

OutOfMemory, OutOufRange

5. Description of the Substring public member functions

The class Substring is just a helper class to get some convenient notation for some string operations. But as this helper class has some public member functions it must be explained here. As the user should not use the class Substring by himself, the constructors of this class are all private. Class String can use them as it is a friend of class Substring.

```
// Here are some sample cases:
String s1 = "1234567";
String s2, s3;
s2 = s1(1,5);         // s2 holds five chars from s1 starting at
                      // position 1 (counted from zero)
s3 = s1.getSubstring(1,5);
assert(s2 == s3);
assert(s2 == "23456");

s3 = s2;
s3.remove(1, 3);
s3.insert(1, "abcd");
s2(1,3) = "abcd";     // deletes "345" from s2 and inserts
                      // "abcd"
assert(s2 == s3);
assert(s2 == "2abcd6");
```

5.1. Assignment

Declarations:

```
void operator=(const String& s)
```

analogous:

```
void operator=(const char* cs)
void assign(const char* cb, SLT n)
void assign(char c, SLT rep = 1)
```

Synopsis:

Assign a String to a Substring. As a Substring will normally only have a pointer to its String, so this operation will change the originally String.

?? Substrings and temporaries ??

Pre-conditions:

None

Post-conditions:

None

Result:

None

Exceptions:

OutOfMemory

References

Plauger, P.J. The Standard C Library. Addison-Wesley 1992.