Document Number:	X3J16/92-0045
	WG21/N0122
Date:	May 20, 1992
Project:	Programming Language C++
Reply To:	Uwe Steinmueller
	uwe.steinmueller@sniap.mchp.sni.de

A String Class in C++

Uwe Steinmueller

SNI AP 44

Siemens Nixdorf Informationssysteme AG 1992

1. Introduction

- Value or pointer semantics for strings. This version assumes value semantics. (Of course, an implemenation may choose to use reference counted pointers, and implement 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 when interfacing to C) we prevent the *implicit* casting to char*.
- National language support (this version only for single byte character sets).
- Storing of ASCII 0x00 an arbitrary positions in a string. We decided to permit storage of 0x00 in a string as it is sometimes needed (otherwise we would be limiting the string class unnecessarily).
- Empirical 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 since a main reason for a standard is to provide 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. In this way 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 a different class.
- A class Stringstream should be provided (not specified here) to supply, for example, the iostream conversion operators.
- Some might argue that the class presented here has far too many features, and many of the extra functions can easily be added by the implementors or the users. But this is exactly the problem: if we know something will be a "common" situation in practice, the standard should guarantee there is an acceptable (what ever this means) solution within 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.

- This proposal removes the indexing member functions (operator[]) and the Substring functions (operator()), as they provide an alias to the internals of the string. The following example gives some motivation for this:

```
// Example
#include <String.h>

String foo(String& s)
{
   s = "";
   return s;
}

char foo1(String& s)
{
   s += "foo";  // or s = "" .....
   return 'a';
}

main()
{
   String s("hallo");

   s[2] = foo1(s);    // memory for s[2] may be invalid
   s(1,2) = foo(s);   // an extra check is needed to insure
                      // that s(1,2) is valid when the string
                      // is assigned to it
}
```

2. Public interface of the string class

The type size_t is used for the string length. We need an extra value to indicate an invalid position (NPOS). This is needed especially for return values of the find operations. It holds that everyvalid position is < NPOS. The class Size_T is used as a wrapper to size_t and used in those cases where we want to guarantee that there is a difference for overloading with an integral type and the size_t type for use in the string class. This is important for constructors with just one argument String::String(size_t) versus String::String(Size_T), the first would be used as a implicit conversion from the integral type of size_t to string (this we do not want) and the second is not.

Names for variables used:

String:	s, s1, s2
Pointer to C string:	cs
Pointer to char* buffer:	cb
size_t:	pos, rep, n
char:	c
ostream:	os
ostream:	is
Size_T:	ic

```
#include <stddef.h>
#include <iostream.h>

extern  const size_t NPOS;      // some value indicating an invalid size_t
typedef int bool;
```

```
class Size_T                    // wrapper for size_t
{
public:
   Size_T(size_t n);
   size_t value();
   ~Size_T();
};
```

why needed ?

```
//
// public interface of String class
//

class String
{
   //
   // Exceptions: OutOfMemory, OutOfRange
   //

public:
   //
   // constructors
   //

   String(Size_T ic = 0);
   String(const String& s);
   String(const char* cb, size_t n = NPOS);
   String(char c, size_t rep = 1);

   //
   // destructor
   //

   ~String();

   //
   // Assignment (value semantics)
   //

   String& operator=(const String& s);

       // needed for convenience and efficiency

       String& operator=(const char* cs);
       String& assign(char c, size_t rep = 1);
       String& assign(const char* cb, size_t n);
       String& operator=(char c);


   //
   // Concatenation
   //

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

       // needed for convenience and efficiency

       String& operator+=(const char* cs);
       String& append(const char* cb, size_t n);
```

```
        String& append(char c, size_t rep = 1);
        String& operator+=(char c);

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

        // needed for convenience and efficiency

        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
    //

    int compare(const String& s);

        // needed for convenience and efficiency

        int compare(const char* cb, size_t n = NPOS);

        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);

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

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

    //
    // Comparison   (depends on collating sequences)
    //

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

    //
    // Insertion at some pos
    //

    String& insert(size_t pos, const String& s);
    // needed for convenience and efficiency

    String& insert(size_t pos, const char* cb, size_t n = NPOS);
    String& insert(size_t pos, char c, size_t rep);

//
// Removal
//

String& remove(size_t pos, size_t n = NPOS);

//
// Replacement at some pos
//

String& replace(size_t pos, size_t n, const String& s);

    // needed for convenience and efficiency

    String& replace(size_t pos, size_t n, const char* cb,
                            size_t l = NPOS);
    String& replace(size_t pos, size_t n, char c, size_t rep);

//
// Subscripting
//

char getAt(size_t pos) const;
void putAt(size_t pos, char c);

//
// Search
//

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

size_t rfind(char c, size_t pos = NPOS) const;
size_t rfind(const String& s, size_t pos = NPOS) const;
size_t rfind(const char* cs, size_t pos = NPOS) const;

//
// Substring
//

String getSubstring(size_t pos, size_t 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');

// ANSI C functionality

// functionality of strpbrk() and strcspn()
```

```
    size_t findFirstOf(String s, size_t pos = 0) const;

    // functionality of strspn()

    size_t findFirstNotOf(String s, size_t pos = 0) const;

    // an equivalent to strtok is not provided, as this should be
    // the task of more powerful special classes (like regex)

    //
    // Miscellanious
    //

    // length

    size_t length() const;

    // copy to C buffer

    size_t copy(char* cb, size_t n, size_t pos = 0) const;

    // get pointer to internal character array

    const char* cStr() const;

    // upper/lower

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

    // Capacity

    size_t reserve();
    void   reserve(Size_T ic);
};
```

3. **Description of the public String member functions**

All member functions which are only declared for the reason of efficiency or convenience are not decribed here as the do not add any functionality.

3.1. Constructors

Declaration:

String(Size_T ic = 0)

Synopsis:

Default constructor creates String of length zero. The implementation can make usage of a capacity value > 0.

Pre-conditions:

None

Post-conditions:

length() == 0

Result:

None

Exceptions:

OutOfMemory

Declaration:

```
String(const String& s)
```

Synopsis:

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

Pre-conditions:

None

Post-conditions:

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

Result:

None

Exceptions:

OutOfMemory

Declaration:

```
String(const char *cb, size_t n = NPOS)
```

Synopsis:

If n == NPOS cb is assumed pointing to a null-terminated C-string and a String containing the characters of this C-string is created.

If n < NPOS a String containing the first n elements of the buffer pointed to by cb is created.

If cs is 0 an empty String is created.

Pre-conditions:

None

Post-conditions:

```
if(cs == 0)
   length() == 0
else if(n == NPOS)
   length() == strlen(cs)
else
   length() == n
memcmp(cStr(), cs, length()) == 0
```

Result:

None

Exceptions:

OutOfMemory

Declaration:

String(char c, size_t rep = 1)

Synopsis:

Creates a String containing rep times character c.

Pre-conditions:

```
rep < NPOS
```

Post-conditions:

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

Result:

None

Exceptions:

OutOfMemory, OutOfRange

3.2. Destructor

Declaration:

```
~String();
```

Synopsis:

Destructs the String and frees all unneeded memory.

Pre-conditions:

None

Post-conditions:

None

Result:

None

Exceptions:

None

3.3. Assignment

Declarations:

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

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(cStr(), s.cStr(), length()) == 0
```

Result:

Reference to String

Exceptions:

OutOfMemory

3.4. Concatenation

Declarations:

String& operator+=(const String& s)

Synopsis:

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

Pre-conditions:

None

Post-conditions:

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

Result:

Reference to String

Exceptions:

OutOfMemory

Declarations:

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

Synopsis:

Concatenate Strings with Strings.

This function is not needed for its functionality.

There might in some cases an unacceptable performance overhead due to creation of temporaries. Speciall care must be taken in the use of the cStr() member functions.

```
  char *p = ("/foo" + '/' + "foo.c").cStr();
  open(p);          // p is not guaranteed 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.cStr(), s1.cStr(), s1.length()) == 0
memcmp(s.cStr() + s1.length(), s2.cStr(), s2.length()) == 0

Result:

String

Exceptions:

OutOfMemory

3.5. 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. Two strings s1 and s2 are assumed to be equal if they have the same length an for all i (0-length-1) s1.getAt(i) == s2.getAt(i) holds. Returns a boolean value.

Pre-conditions:

None

Post-conditions:

None

Result:

Bool

Exceptions:

None

3.6. Comparison

Declaration:

friend int compare(const String& s2)

Synopsis:

Compares String s1 with the String s2. The result should be the same as if the C function memcmp() is performed on the internal representation. It returns an integer less than, equal to, or greater than 0, according as s1 is lexicographically less than, equal to, or greater than s2.

Pre-conditions:

None

Post-conditions:

s.compare(s) == 0

Result:

int

Exceptions:

None

Declarations:

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

analogous:

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)

Synopsis:

Returns the result of the lexicographical compare of the Strings s1 and s2. (using compare() s.o.)

Pre-conditions:

None

Post-conditions:

None

Result:

Bool

Exceptions:

None

3.7. Comparison depending on the status of locale category LC_COLLATE

Declaration:

int strcoll(const String& s)

Synopsis:

Returns the result of strcoll(this->cStr(), s.cStr()). Both Strings are treated like null terminated C-strings. The implementation has to guarantee that both strings can be handled this way (see cStr()). 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(cStr(), s.cStr())

Result:

Bool

Exceptions:

OutOfMemory

Declaration:

String strxfrm()

Synopsis:

Returns a String which is a transformed version of the target String such 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.cStr(), ts2.cStr());
assert(s1.strcoll(s2) == strcoll(s1.cStr(), s2.cStr());
```

Pre-conditions:

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

Post-conditions:

None

Result:

String

Exceptions:

OutOfMemory

3.8. Insert operations

Declarations:

String& insert(size_t pos, const String& s)

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(cStr() + pos, s.cStr(), s.length()) == 0

Result:

Reference to String

Exceptions:

OutOfMemory, OutOfRange

3.9. Removal

Declaration:

String& remove(size_t pos, size_t n = NPOS)

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() == oldlength() - len

Result:

Reference to String

Exceptions:

OutOfMemory, OutOfRange

3.10. Replace operations

Declarations:

String& replace(size_t pos, size_t n, const String& s)

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 convenient.

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

3.11. Subscripting

Declarations:

char getAt(size_t pos) const
void putAt(size_t pos, char c)

Synopsis:

If pos is not a valid position an OutOfRange exception is thrown. The member function getAt returns the character at position pos and putAt sets the character at pos to c.

Pre-conditions:

pos < length()

Post-conditions:

putAt: getAt(pos) == c

Result:

getAt (char) and putAt (void)

Exceptions:

OutOfRange

3.12. Find operations

Declarations:

size_t find(char c, size_t pos = 0) const
size_t rfind(char c, size_t 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 for object is found, the position is returned, or otherwise the value

NPOS.

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

Pre-conditions:

pos != NPOS (for all find versions)

Post-conditions:

None

Result:

size_t

Exceptions:

None (OutOfMemory depending on the search algorithms used)

3.13. Substring

Declarations:

String getSubstring(size_t pos, size_t n) const

Synopsis:

The getSubstring member function 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.

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

Pre-conditions:

pos < length();

Post-conditions:

None

Result:

String

Exceptions:

OutOfRange, OutOfMemory

3.14. 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 = '\n')

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 (without putting any 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). The character c is consumed.

Pre-conditions:

A valid stream

Post-conditions:

None

Result:

ostream& (istream&)

Exceptions:

(see exceptions of the iostream library)

3.15. ANSI C functionality

Declaration:

size_t findFirstOf(const String& s, size_t pos = 0) const

Synopsis:

Returns the position of the first character which is contained in s or NPOS if not found.

Pre-conditions:

pos < length()

Post-conditions:

```
if(result != NPOS)
  s.find(getAt(result)) != NPOS
```

Result:

size_t

Exceptions:

None

Declaration:

size_t findFirstNotOf(const String& s, size_t pos = 0) const

Synopsis:

Returns the position of the first character which is not contained in s or NPOS if not found.

Pre-conditions:

pos < length()

Post-conditions:

```
if(result != NPOS)
  s.find(getAt(result)) == NPOS
```

Result:

size_t

Exceptions:

None

3.16. Miscellanious

Declaration:

size_t length() const

Synopsis:

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

Pre-conditions:

None

Post-conditions:

None

Result:

size_t

Exceptions:

None

Declaration:

size_t copy(char* pc, size_t n, size_t 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:

size_t

Exceptions:

OutOfRange

Declaration:

const char* cStr() 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 member function to get a unique copy of the string. A cast to char* (casting constness away) should not be used, the subscripting functions are 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:

size_t reserve()
size_t reserve(Size_T ic)

Synopsis:

The reserve() member function returns a value which is determined 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:

size_t

Exceptions:

OutOfMemory, OutOufRange

References

Plauger, P.J. The Standard C Library. Eddison Wesley 1992.