A Flexible String Class For C++

Roland J. Schemers III
Computer Science Department
Oakland University

February 14, 1991

Abstract

This paper describes the design and implentation of a flexible string class for C++. It presents the reasoning behind design decisions and gives examples of how to use the class. The project's goals were the following:

1. Strings should be completely dynamic, and grow to meet the user's needs.
2. Strings should be compatible with existing UNIX system calls and C libraries.
3. The string class should support substrings, and the substring class should be derived from the string class.
4. Other classes should be able to inherit the string class easily, and be notified by the string class when a string's value is updated.
5. The user should be able to design classes and functions that work transparently with the existing string class functions.

It is our belief that these design goals were met.

1 Introduction

In working on projects with C++, it was apparent that a string class would greatly reduce the amount of effort needed to write programs. C++ comes with no 'standard' string handling library, and thus every user must write their own. We wanted to write a string class that was fairly complete with the number of built-in operations, and the ease of which it could be inherited by other classes. By including a large number of built-in functions, it was hoped that the user would be able to use the class, and not spend time writing their own functions. Also, the existing functions can easily be used to come up with new functions, with minimal effort spent by the user of the class. For those times when the user does need to add functionality or inherit the string class, virtual functions can be defined to control the action of the string class. It is very simple to write

```
class String {
    char *str; // the actual string
public:
    String() { str = new char; *str='0'; }
    String(const char *s) {
         str = new char[strlen(s)+1]; strcpy(str,s);
    }
    String(const String &s) {
         delete str;
         str = new char[strlen(s)+1]; strcpy(str,s);
    }
    ~String() { delete str; }
    operator < (String &s1) { return strcmp(str,s1.str) < 0; }
    // other relational operators...
    operator char *() { return str; }
    int length() { return strlen(str); }
};
```

Figure 1: A Simple String Class

a trivial string class, and most people end up writing their own. See Figure 1 for an example of a simple string class.

The problem with string classes like the one in Figure 1 is that they are simply encapsulating the standard C way of doing strings, and have the same problems as the standard C strings. These problems all stem from using a **NULL** [4] as a terminator or *sentinel* value to mark the end of the string:

1. Embedded **NULL**s within strings are difficult (if not impossible) to deal with.
2. Calculating the length of the string can be a time consuming task, and is directly related to the length of the string.
3. Trying to keep track of the length of the string in a separate variable can lead to inconsistencies when it is incorrectly maintained.
4. Functions that rely on the **NULL** being there do not act gracefully if it is missing.[1]

Our string class was designed to fix these problems, while remaining highly compatible with existing C libraries.

[1] The words **core dump** come to mind.

```
typedef unsigned short StringRange;

typedef struct {
    StringRange    len;   // length of string (not including NULL)
    StringRange   size;   // allocated size of string
    char         *data;   // pointer to data
} StringData;
```

Figure 2: Data Structure for Representing a String

2 Design

In order to satisfy the design goals of this project, we came up with the data structure shown in Figure 2. As shown by the typedef of StringRange, strings can have a maximum length of 65534 characters (2^{16} - 1 - 1 (for **NULL**) = 65534). If a larger or smaller range is desired, the typedef can be changed and the library can be recompiled. In everyday use a 65534 character maximum should be more than enough. Here is a description of each of the fields in the StringData structure:

len Is the actual length of the string, not including the **NULL**.

size Is the total amount of memory currently allocated.

data Is a pointer to the allocated memory for the string.

We will now describe how design goals were met.

2.1 Strings Should Be Dynamic

The StringData structure holds the total amount of memory currently allocated for a string. When an attempt is made to change the value of a string such that it would be larger than this amount, a new hunk of memory is allocated and the size of the string is increased. Memory size is always increased in multiples of the private static class varible String::hunksize; doing this helps keep resizing of the string to a minimum.

The value of String::hunksize can be examined with the public static class function get_hunksize, which returns the current hunk size. This value can also be set with the public static class function set_hunksize. Setting this value to 1 ensures no internal fragmentation, but increases the frequency of which the string may need to be resized. Setting this value to a large number increases internal fragmentation, but cuts down on the number of times the string may need to be resized. The default value of String::hunksize is 16.

2.2 Strings Should Be Compatible With Standard C Strings

All of our string functions correctly maintain a **NULL** at the end of the string. This ensures maximal compatibility with existing UNIX systems calls and C library functions. Although all the functions maintain this **NULL** internally, they do not use or look at it in any way. This means if the user uses only the supplied string functions, then they can have embedded **NULL**s in their data, at the expense of losing compatiblity with C.

Substrings are also incompatible with functions that rely on the **NULL**. This is because substrings can point anywhere within a string, and thus aren't necessarily **NULL** terminated.

2.3 Substrings

When we started this project, one of the goals was to design the substring class such that it was derived from the string class. Substring classes that aren't inherited from the base string class are usually not treated as equals with strings. For example, you can't have a substring of a substring, or some string functions cannot be called with a substring. We decided this treament was unnecessary, and with careful design of both the string and substring classes can be avoided all altogether.

Our substring class is derived from the string class, and thus is represented by the same data structure as strings. Two additional pointers are used:

- **str** is a pointer to the parent string.
- **sstr** is a pointer to the parent substring, if this is a substring of a substring.

When a substring is created, its data pointer points within the original string to the start of the substring. The length is set to the length of the substring, and the size of the substring is also set to the length. The size field for substrings is not used internally, but is provided for consistency with the String class. When taking the substring of a string, the **str** field points to the parent string, and the **sstr** field is set to **NULL**. When taking the substring of a substring, the **str** field is copied, and the **sstr** is set to point to the parent substring. The **sstr** field thus sets up a linked list of substrings, with the children pointing back to their parents. This is so when a substring is assigned a new value, the **assign** function can traverse this list and update all the parent substrings. When a substring's destructor is called, the data pointer is set to **NULL**, so that when the inherited string destructor is called, the data is not inadvertently deleted.

Since the substring class is derived from the string class, string functions must behave when they are passed a substring. To achieve this two string functions were declared as virtual: **substr** and **assign**. The **substr** function creates a new substring, and the **assign** function assigns a new value to a string or substring.

Just about every single function uses the **substr** function to do its work. For example:

- To insert characters into a string, simply make an assignment to the substring of length 0 at the given position: `s.substr(pos,0) = "hello".`
- To prepend characters to a string, simply make an assignment to the substring of length 0 at position 0: `s.substr(0,0) = "hello".`
- To append characters to a string, simply make an assignment to the substring of length 0 at the end of the string: `s.substr(s.length(),0) = "hello".`
- To remove characters from a string, simply make an assignment to the substring of length n at the given position: `s.substr(pos,n) = "".`

As shown above, almost every function can be broken down as an assignment to a substring. In order for these functions to work, the **substr** function must be declared as virtual. When a substring of a string is taken, the substring must be created so it points back to the parent string. When a substring of a substring is taken, the new substring must point back to the parent string of the old substring. The new substring must also point back to its parent substring.

The **assign** function is also virtual because when a substring is passed to a function that requires a string, that function must call the correct **assign** function when the string is assigned a new value. Also, when changing the value of the substring of a substring, it must correctly update all the parent substrings.

As described above, almost all the functions depend on both the **substr** and the **assign** function[2]. Because of this fact, the substring **assign** function has been optimized to recognize special cases like prepending, inserting, removing, and appending strings.

2.4 Inheritance

When creating new classes from the base string class, users need a way to find out when a string's value has changed. This is achieved through the virtual function **update**. By defining the **update** function in the derived class, the user will be notified whenever a string's value has been changed. Figure 3 shows how to create a class called UpStr, which uses the **update** function to uppercase the string when its value changes. When a user declares an object of type UpStr, it can be used with all the existing string functions. When one of those functions changes the value of the string, the **UpStr::update** function will be called. This function will then correctly change any lowercase letters to uppercase.

[2]Since the substring created with the **substr** function is assigned a new value!

```
class UpStr : public String {
public:
  UpStr()                           : String()    { update(); }
  UpStr(int n)                      : String(n)   { update(); }
  UpStr(char ch)                    : String(ch)  { update(); }
  UpStr(const char *s)              : String(s)   { update(); }
  UpStr(const char *s,int n)        : String(s,n) { update(); }
  UpStr(const String &s)            : String(s)   { update(); }
  UpStr &operator=(char ch)         { (String&) *this = ch; return *this; }
  UpStr &operator=(const char *s)   { (String&) *this = s;  return *this; }
  UpStr &operator=(const String &s) { (String&) *this = s;  return *this; }
  virtual void update();
};

void UpStr::update()
{
int l=length();
char *p=cptr();
  while(l--) {
    if (islower(*p)) *p=_toupper(*p);
    p++;
  }
}

test()
{
 UpStr str("Hello");  // str =="HELLO"

 str +=" world";      // str =="HELLO WORLD"
}
```

Figure 3: Using the **update** Function

2.5 Adding Functionality to Existing String Functions

In order to add functionality to existing string functions, all functions that perform searches can be called with a StringSearch object. By doing this users can create their own special search functions and have them called from within a string function. User-defined functions must conform to the folllowing convention:

1. The search function should be declared as taking two arguments, the first a reference to a constant string, and the second a reference to an integer. It should return an integer value.

2. The search function should perform its search on the string, and return the position within the string the match occurred, or return a -1 if a match was not made.

3. The search function should set its second argument equal to the length of the matched string, or 0 in the case of no match. Note that a 0 length match is valid if the search function does not return a -1.

There are basically two ways to use the StringSearch class. The first is to declare a StringSearch object. Using this method, you pass the address of your string search function to the StringSearch constructor. This method is useful if you don't need all the baggage that comes along with a derived class. The second method is to derive a class from the StringSearch class. In this case the virtual function **search** should be redeclared in the derived class. For example, you could derive a symbol table class from the StringSearch class, and then the string functions would use your symbol table to look up keywords. See Appendix B for examples on how to use the StringSearch class. Here are some of the built-in StringSearch functions:

- **SSwhite** matches 1 or more whitespace characters.
- **SSoptwhite** matches 0 or more whitespace characters.
- **SSnonwhite** matches 1 or more non-whitespace characters.
- **SSint** matches a signed integer value.
- **SSalpha** matches a letter.
- **SSalphanum** matches a letter or a number.
- **SSupper** matches an uppercase letter.
- **SSlower** matches an lowercase letter.
- **SSstr** matches a quoted string.

3 Further Work

Although fairly complete, some issues still remain. Currently strings don't have a pointer to their substrings. If you pass the substring of a global string to a function, then modify this global string within that function, the passed substring would become invalid. This can be corrected and implemented efficiently by having a doubly linked list of substrings within each string. When a string is modified, it must then check this list to see if it has any substrings. If so, it must traverse this list and update them. Another problem is passing two or more substrings of one string to a function. Modifying one of the substrings can leave the other one invalid. This problem can be solved the same as the first. When a substring is modified, it notifies the parent string and then the parent updates all the other substrings.

4 Conclusions

In the end we found that by deriving the substring class from the string class, we were able to re-use over 190 functions from the string class. Only 6 functions from the string class had to be re-defined in the substring class. Also included in the string library is a regular expression class which is derived from the StringSearch class, a string iterator class, and a class for scanning through strings. See the Appendix for examples of these classes.

The string class turned out to be an extremely useful class. We have used it in designing an account registration system for UNIX (4000 lines), and a VMS like help system for UNIX (500 lines). In the registration system, classes such as **UserName** and **ClassName** were derived from the string class. These classes differ from strings in that only certain characters are legal within a user name or a class name. By redefining the virtual function **update** in these classes, checks can be made to ensure these strings contain only legal values. In both of these programs, the string class allowed us to focus on the project and not worry about string manipulation problems. By using this class, a major source of bugs in both C and C++ programs (string handling) was removed.

References

[1] Dewhurst, S.C., and Stark, K.T.: *Programming in C++*. Prentice Hall, 1990.

[2] Ellis, Margaret A., and Stroustrup, Bjarne: *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

[3] Hansen, Tony L.: *The C++ Answer Book*. Addison-Wesly, 1990.

[4] Kernighan, B.W., and Ritchie, D.M.: *The C Programming Language*. Prentice-Hall, 1978.

[5] Lea, Douglas: *libg++, The GNU C++ Library*. USENIX C++ Conference Proceedings, 1988.

[6] Lippman, Stanley B.: *C++ Primer*. Addison-Wesly, 1989.

[7] Stroustrup, Bjarne: *The C++ Programming Language*. Addison-Wesley, 1986.

Appendix

A String Functions

In the following tables assume:

- S is a String
- ch is a char
- n,i,p,l are int
- s is a side type (Left,Right,Both)
- C is either a char, const char *, or const String &
- X is either a C or a const StringSearch &
- Y is either an X or an int (pos)
- Z is either an Y or an int,int (position and length)
- SS is a const StringSearch &
- bool is an int value where zero is false and non-zero is true
- StringRange is a typedef (currently unsigned short)
- String & is a reference to S
- SubString is a SubString within S, and can be used on the left hand of assignment i.e. `S.after("hello")="good bye";`

String Constructors	
Constructor	Description
String()	Empty String
String(int n)	Empty String with a pre-allocated size n
String(char ch)	Initializes String to character ch
String(const char *s)	Initializes String to const char * (**NULL** terminated)
String(const char *s,int n)	Initializes String to const char * of length n
String(const String &s)	Initializes String to another String

String Operators		
Operator	Returns	Description
$S = C$	String&	assigns X to S
$S+ = C$	String&	appends Y to the end of S
$S- = Y$	String&	removes Y if at end of S
$S* = n$	String&	multiplies S by n
$S/ = X$	String&	remove all occurances of X from S
$C1 + C2$	String	concats C1 to C1
$C1 - C2$	String	removes string C2 if at the end of C1
$C1 * n$	String	multiplies C1 by n
$C1/X$	String	remove all occurances of X from C1
$!S$	bool	true if length=0
$C1 < C2$	bool	standard relational
$C1 <= C2$	bool	standard relational
$C1 == C2$	bool	standard relational
$C1! = C2$	bool	standard relational
$C1 >= C2$	bool	standard relational
$C1 > C2$	bool	standard relational
$S[i]$	char	returns char indexed by **i**
$S(Z)$	SubString	returns the substring Z in S
const char *	const char *	converts string to char *
void *	void *	true if length!=0 // if (S) ...

String Functions		
Function	Returns	Description
S.length()	StringRange	returns length of string
S.empty()	bool	true if length=0
S.index(X)	int	position of X within S, or –1
S.contains(X)	bool	true if S contains X
S.substr(p)	SubString	substring starting at p to end of S
S.substr(p,l)	SubString	substring starting at p with length l
S.left(n)	SubString	left n characters in S
S.right(n)	SubString	right n characters in S
S.between(n1,n2)	SubString	characters between position n1 and n2
S.insert(p,C)	String&	inserts C into S at position p
S.prepend(C)	String&	prepends C to S
S.append(C)	String&	appends C to S
S.remove(Z)	String&	removes Z from S
S.before(Y)	SubString	substring in S before Y
S.through(Y)	SubString	substring in S up to and including Y
S.at(Z)	SubString	substring in S at Z
S.from(Y)	SubString	substring in S from Y to end of S
S.after(Y)	SubString	substring in S after Y to end of S
S.except(Z)	String	everything in S except Z
S.pos(p)	String&	assigns p current position within S
S.replace(X1,X2)	String&	replaces X1 with X2
S.split(R,S1[],n)	int	splits S into array S1 on R and returns number split
S.icase()	String&	ignores case during next function
S.ucase()	String&	uses case during next function
S.cptr()	char *	converts string to char *

Special Search Functions		
Function	Returns	Description
S.skip(X)	SubString	skips optional X, returns substring after
S.ws()	SubString	skips whitespace, returns substring after
S.moveto(X)	SubString	moves up to X and returns from X to end
S.moveto(X,p)	SubString	as above, also assigns the position of X to p
S.moveto(X,p,S1)	SubString	as above, also assigns S1 to matched string
S.moveto(X,S1,p,l)	SubString	as above, also assigns l to matched length
S.find(X)	SubString	finds X in S, returns after X to end
S.find(X,p)	SubString	as above, also assigns position of X to p
S.find(X,p,S1)	SubString	as above, also assigns S1 to matched strings
S.find(X,S1,p,l)	SubString	as above, also assigns l to matched length
S.match(X)	SubString	matches X if at leftmost side of S, returns substring after
S.match(X,p)	SubString	as above, also assigns position of X to p
S.match(X,p,S1)	SubString	as above, also assigns S1 to matched string
S.match(X,S1,p,l)	SubString	as above, also assigns l to matched length

String IO Functions		
Function	Returns	Description
ios << S	ostream&	outputs S on stream ios
ios >> S	istream&	inputs S from stream ios
getline(ios,S,ch)	int	reads into S using ch as delimeter and returns length

Misc String Functions		
Function	Returns	Description
S.trim(s)	String&	trims whitespace on side s
S.pad(n,s,ch)	String&	pads to length n with char ch on side s
S.trunc(n)	String&	truncates S to length n
S.upper()	String&	uppercases S
S.lower()	String&	lowercases S
S.reverse()	String&	reverses S

B StringSearch Examples

```
class SSwhitespace : public StringSearch {
public:
     SSwhitespace(int n){ num=n; }
     int search(const String &s, int &matchlen) const ;
private:
     int num;
};

int SSwhitespace::search(const String &s, int &len) const
{
len=0;
int p1;
StringIterator next(s);
char ch;

   while (next(ch))
       if (isspace(ch)) {
            p1=next.pos();
            while (next(ch) && isspace(ch) && (next.pos()-p1)<num);
            len=next.pos()-p1;
            if (len<num) break; // not found
            return p1;
       }
    return -1;

}

int SearchInt(const String &s, int &len) const
{
len=0;
int pos=0,p1;
StringIterator next(s);
char ch;

   while (next(ch)) if (isdigit(ch) || ch=='-') {
      p1=next.pos();
      while (next(ch) && isdigit(ch));
      len=next.pos()-p1;
      return p1;
   }
```

```
   return -1;

}

const SSwhitespace whitespace(5); // find five white space characters
const StringSearch Sint(SearchInt);

void main()
{
String s1("This       is a test");

s1.at(whitespace)=""; // s1=="This  is a test"

s1="this is 1234, not abc";
s1.at(SSint)="one two three four";

}
```

C String Examples

```
String s1,s2,r[10];

s1="12 ab 56";
s1.after("12").before("56") = "34";
// s1=="123456"

s1="12 y 34";
s1.at("y").append("z").prepend("x").insert(0,w);
// s1=="12 wxyz 34"

s1="ab 12 cd";
s1.at("12") += "34";
// s1=="ab 1234 cd"

s1="ab cd ef";
int num=s1.split(SSnonwhite,r,10);
// num==3, r[0]=="ab", r[1]=="cd", r[2]=="ef"

s1="12 34 56";
s1 /= SSwhite;
//s1=="123456"

s1="test.c";
s2=s1-".c";
// s2=="test"

s1="hello";
if (s1.icase()=="HELLO") ... // would be true

s1="show proc/id=12";
s1.ws().find("proc").match("/id=").match(SSint,s2);
// s2=="12"
```

D Regular Expression Examples

```
String s1("123abababaa456");
Regex RXab("[ab]+");

s1.at(RXab)="";
// s1=="123456"
```

```
Regex RXint("-?[0-9]+");
s1="this is 10, not 11";
s1.at("RXint)="ten";
// s1=="this is ten, not 11"
```

E StringScan Examples

```
String s1("these are words")
String w;
StringScan Word(s1,StringScan::ByMatch,SSnonwhite);

while (Word(w)) cout << w << endl;  // output each word in s1

String f;

s1="joe::100:45:Joe Smith:/usr/joe:/bin/seashell");
StringScan Field(s1,StringScan::ByField,String(":"));

Field(f); // f=="joe"
Field(f); // f==""
Field(f); // f=="100"
Field(f); // f=="45"
Field(f); // f=="Joe Smith"
Field(f); // f=="/usr/joe"
Field(f); // f=="/bin/seashell"

String num;
s1="ab 12 cd 34 ef";
StringScan Int(s1,StringScan::ByMatch,SSint);

Int(num); // num=="12"
Int(num); // num=="34"
Int(num); // num==""
```