

Re: A string collection abstract data type

Source: http://coding.derkeiler.com/Archive/C_CPP/comp.lang.c/2007-10/msg04070.html

- *From:* "Charlie Gordon" <news@xxxxxxxxxxxx>
 - *Date:* Mon, 29 Oct 2007 02:32:50 +0100
-

"jacob navia" <jacob@xxxxxxxxxx> a écrit dans le message de news:
4724e895\$0\$27378\$ba4acef3@xxxxxxxxxxxxxxxxxxxx

<original post snipped>

Thank you for your contribution Jacob.
The code was flushed left probably because of tabs, I reformated it and fixed a number of small and not so small issues. See below:

Problems fixed:

- added API remarks
- fixed missing #endif
- fixed indentation and spacing
- used typedef instead of struct tag in implementation
- added const on unmodified string parameters
- #include <stddef.h> instead of <string.h>
- fixed some bogus bit tests: SC->flags * SC_READONLY ?
- added definition for DEFAULT_START_SIZE, allowed it to be 0
- removed redundant initialization of result->count
- used safer MALLOC idiom (à la c.l.c)
- fixed off by one error in ReplaceAt
- fixed memory leak in Resize
- fixed crash on NULL strings in Contains and IndexOf
- used same semantics and method in IndexOf and Contains and shared code
- fixed two off by one errors in InsertAt
- simplified InsertAt
- removed test for condition never reached in RemoveAt
- fixed off by one bug in RemoveAt
- make SetCapacity work for a non empty collection
- added test for MALLOC failure in SetCapacity
- duplicate string in Insert, InsertAt, ReplaceAt; handle NULL and failure
- allow ReplaceAt with an index of SC->count
- used int instead of size_t for count and size for consistency with API.
- improved PrintStringCollection and test main output
- added missing return 0 in main

Here is the corrected version:

Re: A string collection abstract data type

/*-----

StringCollection by Jacob Navia

jacob navia

jacob at jacob point remcomp point fr

logiciels/informatique

<http://www.cs.virginia.edu/~lcc-win32>

Abstract:

Continuing the discussion about abstract data types, in this discussion group, a string collection data type is presented, patterned after the collection in C# and similar languages (Java). It stores character strings, and resizes itself to accommodate new strings when needed.

Interface:

The data structure uses a table of function pointers to provide an extensible basis for many functions without cluttering the user's workspace with too many names.

This architecture is not only extensible at the API level, but it allows "subclassing". If the user is unsatisfied with some of the functions of the API, he/she can:

- o Replace one or more of the function pointers in the table with a function of his/her own.
- o Store somewhere the old function pointer, and replace it with a function of his own that after doing some work calls the original function pointer, optionally doing some work after the original function returns.

This type of extensibility can only be achieved with function pointers. The basic problem with many of the proposals presented here is that they present too much names that can conflict with user names.

This hiding of names is useful in another way, since it allows the API to use completely generic names like "Add" or similar without any ambiguity.

Since those generic names can be used in *other* abstract data types that use the same type of structure, the user code can be made truly general, and it is easy to change from a string collection to a list without too much trouble.

The objective would be to make a standard way of naming things within a container, so that user code is shielded from change when passing from

Re: A string collection abstract data type

Re: A string collection abstract data type

one container to another.

The name of the table of functions member is "lpVtbl" following an old function naming convention under the windows OS and in the COM system. It can be changed of course.

API remarks:

Strings passed to API functions are not modified.

Strings inserted into the collection are duplicated will MALLOC and freed with FREE

Collection can contain NULL strings

The interface is described in the header file <strcollection.h>

-----*/

```
//#include <stddef.h> // for size_t, if needed
```

```
// Forward declaration of the string collection type
```

```
typedef struct _StringCollection StringCollection;
```

```
typedef struct {
```

```
// Returns the number of elements stored
```

```
int (*GetCount)(StringCollection *SC);
```

```
// Is this collection read only?
```

```
int (*IsReadOnly)(StringCollection *SC);
```

```
// Sets this collection read-only or unsets the read-only flag
```

```
int (*SetReadOnly)(StringCollection *SC, int flag);
```

```
// Adds one element at the end. Given string is copied
```

```
int (*Add)(StringCollection *SC, const char *newval);
```

```
// Adds a NULL terminated table of strings
```

```
int (*AddRange)(StringCollection *SC, const char * const *newvalues);
```

```
// Clears all data and frees the memory
```

```
int (*Clear)(StringCollection *SC);
```

```
//Case sensitive search of a character string in the data
```

```
int (*Contains)(StringCollection *SC, const char *str);
```

```
// Copies all strings into a NULL terminated vector
```

```
char **(*CopyTo)(StringCollection *SC);
```

```
//Returns the index of the given string or -1 if not found
```

```
int (*IndexOf)(StringCollection *SC, const char *SearchedString);
```

```
// Inserts a string at the position zero.
```

```
int (*Insert)(StringCollection *SC, const char *str);
```

Re: A string collection abstract data type

Re: A string collection abstract data type

```
// Inserts a string at the given position
int (*InsertAt)(StringCollection *SC, int idx, const char *newval);

// Returns the string at the given position
char *(*IndexAt)(StringCollection *SC, int idx);

// Removes the given string if found
int (*Remove)(StringCollection *SC, const char *str);

//Removes the string at the indicated position
int (*RemoveAt)(StringCollection *SC, int idx);

// Frees the memory used by the collection
int (*Finalize)(StringCollection *SC);

// Returns the current capacity of the collection
int (*GetCapacity)(StringCollection *SC);

// Sets the capacity if there are no items in the collection
int (*SetCapacity)(StringCollection *SC, int newCapacity);

// Calls the given function for all strings.
// "Arg" is a user supplied argument (that can be NULL)
// which is passed to the function to call
void (*Apply)(StringCollection *SC,
int (*Applyfn)(char *, void *arg), void *arg);

// Calls the given function for each string and saves
// all results in an integer vector
int *(*Map)(StringCollection *SC, int (*Applyfn)(char *));

// Pushes a string, using the collection as a stack
int (*Push)(StringCollection *SC, const char *str);

// Pops the last string off the collection
char * (*Pop)(StringCollection *SC);

// Replaces the character string at the given position
// with a copy of a new one
char *(*ReplaceAt)(StringCollection *SC, int idx,
const char *newval);

} StringCollectionFunctions;

// Definition of the String Collection type
struct _StringCollection {
StringCollectionFunctions *lpVtbl; // The table of functions
int count; /* in element size units */
char **contents; /* The contents of the collection */
int capacity; /* in element_size units */
```

Re: A string collection abstract data type

Re: A string collection abstract data type

```
unsigned int flags; // Read-only or other flags  
};
```

```
// This is the only exported function from this module  
StringCollection * newStringCollection(int startsize);
```

```
/*-----end of stringcollection.h
```

Implementation

I haven't had the time to comment this more in depth. I hope that this will be done in the ensuing discussion.

-----stringcollection.c */

```
#include <string.h>  
#include "strcollection.h"
```

```
#ifndef MALLOC  
#include <stdlib.h>  
#define MALLOC(s) malloc(s)  
#define FREE(p) free(p)  
#endif
```

```
// Forward definitions
```

```
static int GetCount(StringCollection *SC);  
static int IsReadOnly(StringCollection *SC);  
static int SetReadOnly(StringCollection *SC, int newval);  
static int Add(StringCollection *SC, const char *newval);  
static int AddRange(StringCollection *SC, const char * const *newvalues);  
static int Clear(StringCollection *SC);  
static int Contains(StringCollection *SC, const char *str);  
static char **CopyTo(StringCollection *SC);  
static int IndexOf(StringCollection *SC, const char *SearchedString);  
static int Insert(StringCollection *SC, const char *str);  
static int InsertAt(StringCollection *SC, int idx, const char *newval);  
static char *IndexAt(StringCollection *SC, int idx);  
static int Remove(StringCollection *SC, const char *str);  
static int RemoveAt(StringCollection *SC, int idx);  
static int Finalize(StringCollection *SC);  
static int GetCapacity(StringCollection *SC);  
static int SetCapacity(StringCollection *SC, int newCapacity);  
static void Apply(StringCollection *SC,  
int(*Applyfn)(char *str, void *arg), void *arg);  
static int *Map(StringCollection *SC, int (*Applyfn)(char *str));  
static int Push(StringCollection *SC, const char *str);  
static char *Pop(StringCollection *SC);  
static char *ReplaceAt(StringCollection *SC, int idx, const char *str);  
static StringCollectionFunctions lpVtableSC = {  
GetCount, IsReadOnly, SetReadOnly, Add, AddRange,
```

Re: A string collection abstract data type

```
Clear, Contains, CopyTo, IndexOf, Insert,  
InsertAt, IndexAt, Remove, RemoveAt,  
Finalize, GetCapacity, SetCapacity, Apply,  
Map, Push, Pop, ReplaceAt,  
};
```

```
static char *DuplicateString(const char *str)  
{  
    char *result;  
  
    if (str == NULL)  
        return NULL;  
    result = MALLOC(strlen(str) + 1);  
    if (result == NULL)  
        return NULL;  
    return strcpy(result, str);  
}
```

```
#define SC_READONLY 1  
#define CHUNKSIZE 20  
#define DEFAULT_START_SIZE 0
```

```
StringCollection *newStringCollection(int startsize)  
{  
    StringCollection *SC = MALLOC(sizeof(*SC));  
  
    if (SC == NULL)  
        return NULL;  
  
    SC->lpVtbl = &lpVtblSC;  
    SC->count = 0;  
    SC->contents = NULL;  
    SC->capacity = 0;  
    SC->flags = 0;  
  
    if (startsize == 0)  
        startsize = DEFAULT_START_SIZE;  
    if (startsize) {  
        SC->capacity = startsize;  
        SC->contents = MALLOC(startsize * sizeof(*SC->contents));  
        if (SC->contents == NULL) {  
            FREE(SC);  
            return NULL;  
        }  
        // useless, for cleanliness only  
        memset(SC->contents, 0, startsize * sizeof(*SC->contents));  
    }  
    return SC;  
}
```

```
static int GetCount(StringCollection *SC)
```

Re: A string collection abstract data type

```
{
return SC->count;
}

static int IsReadOnly(StringCollection *SC)
{
return (SC->flags & SC_READONLY) ? 1 : 0;
}

static int SetReadOnly(StringCollection *SC, int newval)
{
int oldval = (SC->flags & SC_READONLY) ? 1 : 0;

if (newval)
SC->flags |= SC_READONLY;
else
SC->flags &= ~SC_READONLY;
return oldval;
}

static int Resize(StringCollection *SC)
{
int newcapacity = SC->capacity + CHUNKSIZE;
char **newcontents;

newcontents = MALLOC(newcapacity * sizeof(*newcontents));
if (newcontents == NULL)
return 0;
memcpy(newcontents, SC->contents, SC->count * sizeof(*newcontents));
// useless, for cleanliness only
memset(newcontents + SC->count * sizeof(*newcontents),
0, (newcapacity - SC->count) * sizeof(*newcontents));
FREE(SC->contents);
SC->contents = newcontents;
SC->capacity = newcapacity;
return 1;
}

static int Add(StringCollection *SC, const char *str)
{
char *newval = NULL;

if (SC->flags & SC_READONLY)
return -1;
if (SC->count >= SC->capacity) {
if (!Resize(SC))
return 0;
}
if (str) {
newval = DuplicateString(str);
if (newval == NULL)
```

Re: A string collection abstract data type

```
return 0;
}
SC->contents[SC->count] = newval;
return ++SC->count;
}

static int AddRange(StringCollection *SC, const char * const *data)
{
int i;

if (SC->flags & SC_READONLY)
return 0;
for (i = 0; data[i] != NULL; i++) {
int r = Add(SC, data[i]);
if (r <= 0)
return r;
}
return SC->count;
}

static int Clear(StringCollection *SC)
{
int oldval = SC->count, i;

if (SC->flags & SC_READONLY)
return 0;
for (i = 0; i < SC->count; i++) {
FREE(SC->contents[i]);
SC->contents[i] = NULL;
}
SC->count = 0;
return oldval;
}

static int Contains(StringCollection *SC, const char *str)
{
return (IndexOf(SC, str) >= 0);
}

static char **CopyTo(StringCollection *SC)
{
char **result = MALLOC((SC->count + 1) * sizeof(*result));
int i;

if (result == NULL)
return NULL;
for (i = 0; i < SC->count; i++) {
// XXX: MALLOC failure ignored
result[i] = DuplicateString(SC->contents[i]);
}
result[i] = NULL;
}
```

Re: A string collection abstract data type

```
return result;
}

#ifdef __LCC__
/* The lcc-win compiler allows operator overloading, and allows using
this abstract data type with the more natural [ ] operators */
char * __declspec(naked) operator[](StringCollection *SC, int idx)
{
}
#endif

static int IndexOf(StringCollection *SC, const char *str)
{
int c, i;

if (str == NULL) {
for (i = 0; i < SC->count; i++) {
if (SC->contents[i] == NULL)
return i;
}
return -1;
} else {
c = *str;
for (i = 0; i < SC->count; i++) {
if (SC->contents[i] == NULL)
continue;
if (c == SC->contents[i][0] && !strcmp(SC->contents[i], str))
return i;
}
return -1;
}
}

static char *IndexAt(StringCollection *SC, int idx)
{
if (idx < 0 || idx >= SC->count)
return NULL;
return SC->contents[idx];
}

static int InsertAt(StringCollection *SC, int idx, const char *str)
{
char *newval = NULL;

if (SC->flags & SC_READONLY)
return 0;
if (idx < 0 || idx > SC->count)
return -1;
if (SC->count >= SC->capacity) {
if (!Resize(SC))
return 0;
}
```

Re: A string collection abstract data type

```
}
if (str) {
newval = DuplicateString(str);
if (newval == NULL)
return 0;
}
if (idx < SC->count) {
memmove(SC->contents + idx + 1, SC->contents + idx,
(SC->count - idx) * sizeof(*SC->contents));
}
SC->contents[idx] = newval;
return ++SC->count;
}
```

```
static int Insert(StringCollection *SC, const char *str)
{
return InsertAt(SC, 0, str);
}
```

```
static int RemoveAt(StringCollection *SC, int idx)
{
if (idx < 0 || idx >= SC->count)
return -1;
FREE(SC->contents[idx]);
memmove(SC->contents + idx, SC->contents + idx + 1,
(SC->count - idx - 1) * sizeof(*SC->contents));
SC->contents[--SC->count] = NULL;
return SC->count;
}
```

```
static int Remove(StringCollection *SC, const char *str)
{
int idx = IndexOf(SC, str);
if (idx < 0)
return idx;
return RemoveAt(SC, idx);
}
```

```
static int Push(StringCollection *SC, const char *str)
{
char *newval = NULL;

if (SC->flags & SC_READONLY)
return 0;
if (SC->count >= SC->capacity) {
if (!Resize(SC))
return 0;
}
if (str) {
newval = DuplicateString(str);
if (newval == NULL)
```

Re: A string collection abstract data type

```
return 0;
}
SC->contents[SC->count++] = newval;
return SC->count;
}

static char * Pop(StringCollection *SC)
{
char *result;

if ((SC->flags & SC_READONLY) || SC->count == 0)
return NULL;
SC->count--;
result = SC->contents[SC->count];
SC->contents[SC->count] = NULL;
return result;
}

static int Finalize(StringCollection *SC)
{
int result = SC->count, i;

for (i = 0; i < SC->count; i++) {
FREE(SC->contents[i]);
}
FREE(SC->contents);
FREE(SC);
return result;
}

static int GetCapacity(StringCollection *SC)
{
return SC->capacity;
}

static int SetCapacity(StringCollection *SC, int newcapacity)
{
char **newcontents;

if (newcapacity < 0)
return 0;

newcontents = MALLOC(newcapacity * sizeof(*newcontents));
if (newcontents == NULL)
return 0;

while (SC->count > newcapacity) {
FREE(SC->contents[--SC->count]);
}
memcpy(newcontents, SC->contents, SC->count * sizeof(*newcontents));
// useless, for cleanliness only
```

Re: A string collection abstract data type

```
memset(newcontents + SC->count * sizeof(*newcontents),
0, (newcapacity - SC->count) * sizeof(*newcontents));
FREE(SC->contents);
SC->contents = newcontents;
SC->capacity = newcapacity;
return 1;
}

static void Apply(StringCollection *SC,
int (*Applyfn)(char *str, void *arg), void *arg)
{
int i;

for (i = 0; i < SC->count; i++) {
Applyfn(SC->contents[i], arg);
}
}

static int *Map(StringCollection *SC, int (*Applyfn)(char *str))
{
int *result = MALLOC(SC->count * sizeof(*result));
int i;

if (result == NULL)
return NULL;
for (i = 0; i < SC->count; i++) {
result[i] = Applyfn(SC->contents[i]);
}
return result;
}

#ifdef __LCC__
/* The lcc-win compiler allows operator overloading, and allows using
this abstract data type with the more natural [=] operators */
char * __declspec(naked) operator[]=(StringCollection *SC,
int idx, char *newval)
{
}
#endif

static char *ReplaceAt(StringCollection *SC, int idx, const char *str)
{
char *newval = NULL;

if (SC->flags & SC_READONLY)
return NULL;
if (idx < 0 || idx > SC->count)
return NULL;
if (idx == SC->count) {
if (!Add(SC, str))
return NULL;
}
```

Re: A string collection abstract data type

```
} else {
if (str) {
newval = DuplicateString(str);
if (newval == NULL)
return NULL;
}
FREE(SC->contents[idx]);
SC->contents[idx] = newval;
}
return SC->contents[idx];
}

#ifdef TEST

#include <stdio.h>

static void PrintStringCollection(StringCollection *SC)
{
int i;

printf("Count %d, Capacity %d {\n",
SC->count, SC->capacity);
for (i = 0; i < SC->count; i++) {
printf(" \"%s\", \n", SC->lpVtbl->IndexAt(SC, i));
}
printf("}\n");
}

int main(void)
{
StringCollection *SC;
int count;
char *p;

SC = newStringCollection(0);
printf("newStringCollection(%d)\n", 0);
PrintStringCollection(SC);
SC->lpVtbl->Finalize(SC);

SC = newStringCollection(10);
printf("newStringCollection(%d)\n", 10);
PrintStringCollection(SC);
printf("Add \"%s\"\n", "Martin");
SC->lpVtbl->Add(SC, "Martin");
printf("Insert \"%s\"\n", "Jacob");
SC->lpVtbl->Insert(SC, "Jacob");
count = SC->lpVtbl->GetCount(SC);
if (count != 2)
printf("Count should be 2, is %d\n", count);
PrintStringCollection(SC);
printf("Insert at %d \"%s\"\n", 1, "Position 1");
```

Re: A string collection abstract data type

```
SC->lpVtbl->InsertAt(SC, 1, "Position 1");
printf("Insert at %d \"%s\"\n", 2, "Position 2");
SC->lpVtbl->InsertAt(SC, 2, "Position 2");
PrintStringCollection(SC);
printf("Remove \"%s\"\n", "Jacob");
SC->lpVtbl->Remove(SC, "Jakob");
PrintStringCollection(SC);
printf("Push \"%s\"\n", "pushed");
SC->lpVtbl->Push(SC, "pushed");
PrintStringCollection(SC);
p = SC->lpVtbl->Pop(SC);
FREE(p);
printf("Pop -> \"%s\"\n", p);
PrintStringCollection(SC);
p = SC->lpVtbl->IndexAt(SC, 1);
printf("IndexAt %d -> \"%s\"\n", 1, p);
PrintStringCollection(SC);
SC->lpVtbl->Finalize(SC);

return 0;
}
#endif

/*-----*/
```

There are some remaining issues:

Return values should be documented... in fact the whole API should be. A concise paragraph before each method would suffice.

We should decide if the collection is indexed with `int` or `size_t`, or some appropriate typedef and make API consistent.

Why does the apply function in `Apply` return `int`? the value is ignored.
Why does the apply function in `Map` not take an extra arg like that of `Apply` does?

Memory allocation could be handled with methods as well, but that's debatable.

"Subclassing" should be done by changing the pointer to the virtual table for specific instances, and make that point to a copy of the original one, with appropriate function pointer changed, and possibly extra methods defined. The way you document affects **all** `StringCollections` in a given program.

Please allow me to insist on you publishing the source your compiler so you can get this kind of help everywhere. Under an appropriate licence, open sourcing it would not prevent you from selling it to businesses for commercial use.

Re: A string collection abstract data type

Good night.

—

Chrlie.

.