

New directions in C

Jacob Navia

1.0 Motivation

1.1 The current situation

The development of the C++ language has had an adverse effect in the development of C. Since C++ was designed as the “better C”, C was (and is) presented as the example of “how not to do things”, even if both languages retained a large common base. Each and every introduction to C++ never misses to point out the flaws of C and why C++ is the solution for all those problems.

The need for a simple and efficient language persists however, and C is the language of choice for many systems running today and many new ones. However, programming in C is made more difficult than it should be because of some glaring deficiencies like its buggy string library, the lack of a common library for the most used containers like lists, stacks, and other popular data structures.

Since C++ went out to be “the better C”, it is important to avoid reintroducing the whole complexities of C++ into C and keep the language as simple as it should be, but not simpler than the minimum necessary to use it without much pain.

Of course the idea of improving C is doomed according to the C++ people, that obviously will say that the solution is not to improve C but to come over to C++. This same failure forecast is found by some C people, that see any change to their baby as the beginning of the end of the “spirit of C”.

1.1.1 The most important changes needed.

This proposal seeks to address the following problems as they appear in our day to day programming.

1. The lack of a counted string data structure, that would give programmers an alternative to the inefficient and extremely error prone “zero terminated” strings.
2. The need to add new numerical types to the language.
3. The need for an abstract “containers” library that would allow to improve the developing of portable programs by providing simple data structures like lists, flexible arrays, stacks, and others, without any of the complexity of the STL of C++.

Let’s see this points in more detail.

1.1.2 A counted string data structure

C has traditionally defined the string data type as a sequence of bytes ending with a binary zero byte. This is an extremely inefficient way of storing strings since the length of the string, one of the most used information about the string, must be recalculated at each

access. This implies in most cases a function call to `strlen()`, or a loop. In case of long strings this makes C less efficient than a BASIC interpreter written in C#.

This will be recognized by most C programmers, and the existence of countless string libraries testifies to this fact. The problem of those libraries is that they are not standard and must be ported from one application to the next, and from one operating system to the next. What is needed is language support for counted strings.

Another requirement is that the syntax for accessing strings and the whole library should be as similar to the traditional C syntax as possible. The goal must be that the code running now should be ported with minimal effort to the new environment, so that the user can switch back to zero terminated strings easily if it must run in an environment where the library is absent.

1.1.3 The need for new numeric types

Currently, there are several proposals for new types of numbers pending with the standards committee.

1. Extension for the programming language C to support decimal floating-point arithmetic. Document ISO/IEC TR 24732.
2. Fixed point arithmetic proposal, in ISO/IEC DTR 18037.
3. The standard doesn't provide any means of providing extended precision integer and floating point numbers. "Bignum" integer packages abound, but since there is no language support, their usage is very cumbersome.

All those different types of numbers and more can be integrated into the language with a uniform method, that provides a standard way of doing this changes, and doesn't divide the language in subsets, where we would have one subset with fixed point arithmetic syntax, and another with decimal arithmetic, and many implementations with none.

1.1.4 The need for an abstract "containers" library

In the current state, C doesn't have any language support (in the form of a standard library) for any data structure like stacks/lists/flexible arrays/ and many others. What is needed is a standard way of accessing those commonly used data structures using a uniform syntax, what would allow programs that use them to be portable. The situation today is similar to the situation with character strings, where there are a lot of libraries, but all incompatible with each other. What is needed is to keep those libraries but to standardize a syntax that would allow user programs to be portable.

1.2 The proposed changes

All proposed changes have no impact in the existing language, and can be used when necessary without having any impact in the performance of the already existing code.

This document then, proposes the development of several enhancements to the language, mostly compatible with their C++ counterparts. The main aim is to make C programming easier, more secure and more flexible than it is now. Each addition is justified by the improvements it brings, and possible uses and mis-uses are discussed.

All this developments are implemented using the lcc-win32 compiler system¹. These are not just proposals but a reference implementation exists, and it is widely distributed since several years.

The main propositions developed here are:

1. Operator overloading
2. Garbage Collection
3. Generic functions
4. Default function arguments
5. References

All this propositions have as a goal increasing the level of abstraction used by C programmers without unduly increasing the complexity of the implementation. All this enhancements have added only about 2 000 lines of code to the original code of the lcc-win compiler. This is extremely small, and proves that apparently difficult extensions can be inserted into an existing compiler without any code bloat.

Each enhancement can be viewed separately, but their strength is only visible when they all work together.

The first part of this document details the specifications for the proposed changes, the second describes applications for them in the form of a string library and a container library. There you see how these enhancements work, and how they could be used to implement a good standard library for C.

1.2.1 Operator overloading

Operator overloading allows the user to define its own functions for performing the basic operations of the language for user defined data types.

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

1.2.2 Motivation

Many languages today accept operator overloading. Among them Ada, C++, C#, D, Delphi, Perl, Python, Visual Basic, Ruby, Smalltalk, Eiffel.

The purpose of this enhancement within the context of the C language is to:

1. Allow the user to define new types of numbers or “numeric” objects.
2. Allow a generic access to containers by allowing the user to define special ‘array like’ access to containers using the overloaded '[' and ']' operators.

1) Many applications need to define special kinds of numbers. Rational arithmetic, “big” numbers, extended precision floating point come immediately to mind, and there are surely many others. For instance the Technical Report 24732 of the ISO/IEC proposes a new kind of decimal floating point, the Technical Report DTR 18037 proposes fixed point operations, etc. All of them propose changes to the language in the form of new keywords. A conceptually simpler solution is to allow a single change that would accommodate all those needs without making C impossible to follow by adding a new keyword for every kind of number that users may need.

2) The usage of arrays in C is peculiar and quite difficult to use. Allowing users to define new kinds of array access permits to integrate many needs like bounds checking within the language without adding any special new syntax. There are several propositions about bounded strings circulating in the standardization committee, and they propose several different enhancements to the existing library, mainly by the addition of several parameters to the string functions to pass the length of the receiving strings. This is a misguided approach since it still leaves too much work to the programmer that should still take care of following the size of each string he/she uses in the program without ever making a mistake. This is asking for trouble.

Obviously counting string lengths is better done by machines. The length should be a quantity associated to the string and managed at runtime by the routines using those strings. This would be, by the way, much more efficient than searching the terminating zero each and every time a string is used.

Still, it is needed to retain the original array-like syntax for this strings or bounded buffers/arrays. It is an intuitive syntax, in use almost in all programming languages and it will allow an easier transition of existing code. Then, we need an overloading of the operator `index []`.

These are the main objectives of this syntax change. Note that it is *not* in the design objectives to replace normal procedures like string concatenation with an overloaded “add” operator. It is obvious too, that once this syntax is in use, such bad applications can be programmed and it is impossible to do anything about them.¹

1. “Adding” strings is a bad idea because `a+b` is different than `b+a`, what is counterintuitive. Besides, a function like “Append” or “Strcat” is much clearer than `a+b`.

1.2.3 Syntax:

result-type operator *symbol* (parameters)

“Result-type” is the type of the operator result.

“symbol” is one of the operator symbols (+ - * / << [], etc. Explained in detail later)

An exception to the above rule are the pre-increment and pre-decrement operators, that are written:

result-type ++operator `(` parameters `)`

result-type --operator `(` parameters `)`

This enhancement doesn’t use any new keywords. The C99 standard explicitly forbids new keywords, and this has been respected. It remains to be seen if really an operator keyword is needed. As implemented in the reference implementation it is still possible to write:

```
int operator = 67;
```

without any problems.

The rules for using the operator identifier are as follows:

1. It must appear at the global level.
2. It must be preceded by a type name.
3. It must be followed by one of the operator symbols, and then an opening parentheses, an argument list that can’t be empty and can’t be longer than 2 arguments, followed by a closing parenthesis.
4. If it is followed by a “;” it is a prototype for an operator defined elsewhere. All rules applying to prototypes apply equally to this prototype.
5. If it is followed by an opening brace it is the beginning of an operator definition. All rules that apply to function definition apply also here.

Note that all this rules are no longer needed if the standard would accept a new “operator” keyword.

The operators that can be overloaded are:

TABLE 1. Operator Symbols

Symbol	Name	Description and prototype
+	plus	Type operator+(const Type arg1, const Type arg2); The arguments aren't necessarily of the same type. Pointers can't be used for arg1 or arg2.
-	minus	Type operator-(const Type arg1, const Type arg2); The parameters can't be pointers.
-	unary_minus	Type operator-(const Type arg1);
*	multiply	Type operator*(const Type arg1, const Type arg2);
/	divide	Type operator/(const Type arg1, const Type arg2);
==	equal	int operator==(const Type arg1, const Type arg2); The parameters can't be pointers and the result type must be an integer
!=	notEqual	int operator!=(const Type arg1, const Type arg2); The parameters can't be pointers, and the result type must be an integer.
++	Post-Increment	Type operator++(Type arg1); The parameter can't be a pointer.
--	Post-decrement	Type operator--(Type arg1); The parameter can't be a pointer
++	Pre-Increment	Type ++operator(Type arg1); The parameter can't be a pointer.
--	Post-Decrement	Type --operatorType arg1); The parameter can't be a pointer.
<	less	int operator<(const Type arg1, const Type arg2); The parameters can't be pointers. Result type is int.
<=	lessequal	int operator<=(const Type arg1, const Type arg2); The parameters can't be pointers. Result type is int.
>=	greaterequal	int operator>=(const Type arg1, const Type arg2); The parameters can't be pointers. Result type integer.
!	logicalNot	int operator!(const Type arg1);
~	not	Type operator~const (Type arg1);
%	mod	Type operator%(const Type arg1, const Type arg2);
<<	leftshift	Type operator<<(const Type arg1, const Type arg2);
>>	rightshift	Type operator>>(const Type arg1, const Type arg2);
=	asgn	Type operator=(const Type1 arg1, const Type2 arg2);
^	xor	Type operator^(const Type arg1, const Type arg2);
&	and	Type operator&(const Type arg1, const Type arg2);
	or	Type operator (const Type arg1, const Type arg2);
[]	index	Type operator[] (Type table, int index);

TABLE 1. Operator Symbols

Symbol	Name	Description and prototype
[] =	indexasgn	Type operator [] = (Type table, int index, Type Newvalue);
+=	plusasgn	Type operator += (Type &arg1, Type arg2); References can be replaced by arrays of length 1.
-=	minusasgn	Type operator -= (Type &arg1, Type arg2);
*=	multasgn	Type operator *= (Type &arg1, Type arg2);
/=	divasgn	Type operator /= (Type &arg1, Type arg2);
<<=	lshasgn	Type operator <<= (Type &arg1, Type arg2);
>>=	rshasgn	Type operator >>= (Type &arg1, Type arg2);
()	cast	Type operator () (Type arg1); The parameter can't be a pointer.
*	indirection	Type operator * (Type arg1); The parameter can't be a pointer.

1.2.4 Rules for the arguments

At least one of the parameters for the overloaded operators must be a user defined type. They must be a structure or a union, not just a typedef for a basic type. Pointers to structures or unions are accepted only when the operator has no standard C counterpart for operations with pointers. Pointer multiplication is not allowed in standard C, so an overloaded multiplication operator that takes two pointers is not ambiguous. Addition of pointer and integer is well defined in C, so an operator “add” that would take a pointer and an integer would introduce an ambiguity in the language, and it is therefore not allowed. The same for pointer subtraction.

The result type of an operator can be any type, except for the equality and the other comparison operators that always return an integer.

The number of arguments are fixed for each operator (as described in the table above). It is not possible to change this and define ternary operators that would make two multiplications, for instance.

When an operator needs to modify its argument (for instance the assignment operator, or the += operator) and can't take pointers, it should take a reference to the object to be modified. This ties somehow this enhancement to the second one described further down, references.

Overloaded operators can't have default arguments.(page 13)

1.2.5 Name resolution

Name resolution is the process of selecting the right operator from a list of possible candidates.

There can be only one operator that applies to a given combination of input arguments, i.e. to a given signature. If at the end of the name resolution procedure more than one overloaded operator is found a fatal diagnostic is issued, and no object file is generated.

Step one: Compare the input arguments for the list of overloaded functions for this operator without any promotion at all. If at the end of this operation one and only one match is found return success.

Step two: Compare input arguments ignoring any signed/unsigned differences. If in the implementation `sizeof(int) == sizeof(long)` consider long and int as equivalent types. The same if in the implementation `sizeof(int) == sizeof(short)`. Consider the enum type as equivalent to the int type. If at the end of this operation one and only one match is found return success.

Step three: Compare input arguments ignoring all differences between numeric arguments. If at the end of this operation only one match is found return success.

Step four: If the operation is one of the comparisons operators (equal, not equal, less, less-equal greater, greater-equal) invert the operation and try to find a match. If it is found invert the order of the arguments for less, less-equal greater-equal greater, or, call the not operator for equal and not equal. It is assumed that:

TABLE 2. Assumed operator equivalences

Operator	Equivalent
equals	! different
different	! equals
less	invert arguments: less(a,b) is equivalent to greater-equal(b,a)
less-equal	invert arguments less-equal(a,b) is equivalent to greater(b,a)
greater-equal	invert arguments: greater-equal(a,b) is equivalent to less(b,a)
greater	invert arguments: greater(a,b) is equivalent to less-equal(b,a)
+=	operator binary add + operator assignment
-=	operator binary minus + operator assignment
... all others	

Step five: Return failure.

1.2.6 Differences to C++

In the C++ language, you can redefine the operators `&&` (and) `||` (or) and `,` (comma). You cannot do this in C. The reasons are very simple.

In C (as in C++), logical expressions within conditional contexts are evaluated from left to right. If, in the context of the AND operator, the first expression returns a FALSE value, the others will NOT be evaluated. This means that once the truth or falsehood of an expression has been determined, evaluation of the expression ceases, even if some parts of the expression haven't yet been examined.

Now, if a user wanted to redefine the operator AND or the operator OR, the compiler would have to generate a function call to the user-defined function, giving it all the arguments of BOTH expressions. To make the function call, the compiler would have to evaluate them both, before passing them to the redefined operator &&.

Consequence: all expressions would be evaluated and expressions that rely on the normal behavior of C would not work. The same reasoning can be applied to the operator OR. It evaluates all expressions, but stops at the first that returns TRUE.

A similar problem appears with the comma operator, which evaluates in sequence all the expressions separated by the comma(s), and returns as the value of the expression the last result evaluated. When passing the arguments to the overloaded function, however, there is no guarantee that the order of evaluation will be from left to right. The C standard does not specify the order for evaluating function arguments. Therefore, this would not work.

Another difference with C++ is that here you can redefine the operator []=, i.e., the assignment to an array is a different operation than the reference of an array member. The reason is simple: the C language always distinguishes between the operator + and the operator +=, the operator * is different from the operator *=, etc. There is no reason why the operator [] should be any different.

This simple fact allows you to do things that are quite impossible for C++ programmers: You can easily distinguish between the assignment and the reference of an array, i.e., you can specialize the operation for each usage. In C++ doing this implies creating a “proxy” object, i.e., a stand-by construct that senses when the program uses it for writing or reading and acts accordingly. This proxy must be defined, created, etc., and it has to redefine all operators to be able to function. In addition, this highly complex solution is not guaranteed to work! The proxies have subtle different behaviors in many situations because they are not the objects they stand for.

In C++ the [] operator can be only defined within a class. There are no classes in C, and the [] operator is defined like any other.

1.2.7 Array Initialization

When an initialization is declared at the function scope level or a higher level, the overloaded operator assignment takes precedence from the normal assignment. For instance, if the program has an overloaded operator assignment:

```
Mytype operator=(Mytype &dst,int n)
{ ... }
```

visible at a given scope, the statement:

```
Mystruct tab[] = {0,1,2,3};
```

will result in four calls to the overloaded operator. Note that this is possible only at the function scope level, not at the global level. This can result in an inconsistency, but this inconsistency is already in the language. You can write in C99:

```
double tab[] = {sqrt(2),sqrt(3)};
```

at function scope or higher, but NOT at the global scope. This proposal doesn't address this inconsistency in the language, what should be part of another discussion.

1.3 Garbage Collection

1.3.1 Motivation

A considerable part of the time we spend programming in C is spent trying to find and fix problems related to memory allocation. The primitives provided by the C runtime "malloc" and "free" are very error prone, the slightest error can have catastrophic consequences, consequences that are not seen immediately but can manifest in other completely unrelated part of the program.

The memory manager (or garbage collector) is obviously not an universal solution. There are applications where its usage is impossible (real time applications, or memory constrained environments) and it brings also a new host of problems. Basically however, its usage is simple and the impact in the syntax of C is zero.

A garbage collector runs in the background or it is called when an allocation is done, to find all pieces of unused memory in the system, and to recycle them by providing it to the user in the next allocation request.

Programming is simplified enormously because it is no longer necessary for the programmer to track each piece of memory he/she uses individually or to design yet another memory manager. Any piece of code can pass a buffer to any other without establishing a protocol for freeing the allocated buffer.

In another context, a garbage collector allows to make allocations within an overloaded operator without having to manage the release of the allocated memory.

1.3.2 Syntax:

No new syntax is necessary, just one more library function:

```
void *GC_malloc(size_t);
```

The implementation can add other memory allocation/release functions, all using the GC_ prefix.

1.3.3 Usage

One of the big problems introduced by the garbage collector is keeping somewhere an unneeded reference to an object that should be discarded, what provokes that is never released, hence, a memory leak. This bugs are extremely difficult to find.

1.4 Generic functions

1.4.1 Motivation

Like an ordinary function, a generic function takes arguments, performs a series of operations, and perhaps returns useful values. An ordinary function has a single body of code that is always executed when the function is called. A generic function has a set of bodies of code of which only one is selected for execution. The selected body of code is determined by the types of the arguments to the generic function.

Ordinary functions and generic functions are called with identical function-call syntax.

Generic functions were introduced into C by the C99 standard with the header `tgmath.h`. This introduction was severely limited to some mathematical functions but pointed to the need of having one name to remember instead of memorizing several for functions that essentially do the same thing but with slightly different data types.

With the proliferation of numeric types in C it is obvious that remembering the name of each `sin` function, the one for floats the one for complex, the one for long doubles, etc. uses too much memory space to be acceptable. Here I am speaking about the really important memory space: human memory space, that is far more precious now that the cheap random access memory that anyone can buy in the next supermarket. As far as I know there are no human memory extensions for sale, and an interface and a programming language is also judged by the number of things the user must learn by heart, i.e. its memory footprint.

To reduce the complexities of C interfaces two solutions are proposed: The first is the use of generic functions, i.e. functions that group several slightly different task into a single conceptual one, and default arguments, that reduce the number of arguments that the user must supply in a function call, and therefore its memory footprint. Default arguments will be explained in the next section.

1.4.2 Syntax

```
result-type overloaded functionName( argument-list )
```

The same rules apply to the identifier “overloaded” as to the identifier “operator” above. It is not a keyword in this implementation to remain compatible with the standard.

The compiler will synthesize a name for this instance of the overloaded function from the name and its argument list.

If this instance of the overloaded function should be an alias for an existing function, the syntax is as follows:

```
result-type someFn( argument-list );
```

```
result-type overloaded functionName.someFn( argument-list )
```

This second form avoids any automatically generated names, what makes the code binary compatible with other compilers.

1.4.3 Usage rules

An overloaded function must be a new identifier. It is an error to declare a function as overloaded after the compiler has seen a different prototype in the same scope.

It is also an error to declare a function that was declared as overloaded as a normal function.

The rules for name resolution are the same as the rules for operator overloading excepting of course the operator equivalence rule.

Generic functions can't have default arguments.(page 13)

1.5 Default arguments

1.5.1 Motivation

Default arguments are formal parameters that are given a value (the default value) to use when the caller does not supply an actual parameter value. Of course, actual parameter values, when given, override the default values.

In many situations, some of the arguments to a function can be default values. This simplifies the interface for the function and at the same time keeps the necessary option for the user of the function, to specify some corner cases exactly.

Default arguments are used in Python, Fortran, the “Mathematica” language, Lisp, Ruby. Tcl/tk, Visual Basic.

1.5.2 Syntax

```
return-type fnName(arg1, arg2, arg3=<expr>, arg4=<expr>);
```

1.5.3 Usage Rules

The expressions used should be constant expressions. All mandatory arguments should come first, then, the optional arguments.

It is an error to redefine a function that has optional arguments into another with a different list of optional arguments or with different values.

Note that is a very bad idea to change the value of a default argument since all code that uses that function depends implicitly in the value being what is declared to be. A change in the value of the default value assigned to an argument changes all calls to it implicitly.

The expression given as default argument will be evaluated within the context of each call. If the expression contains references to global variables their current value will be used.

1.6 References

1.6.1 Motivation

References are pointers to specific objects. They are immediately initialized to the object they will point to, and they will never point to any other object. They are immediately dereferenced when used.

This addition is necessary for the overloaded operators that modify their arguments and can't receive pointers.

References can't be reassigned, and pointer arithmetic is not possible with them.

2.0 Containers

2.1 Abstract data types

Abstract data types are abstractions from a data structure that consists of two parts: a *specification* and an *implementation*. The specification defines how the data is stored without any implementation details, a set of operations that the user can invoke on the stored data, and the error handling within the context of each operation.

For a single implementation of an abstract data type there can be many possible implementations. This implementations can be changed at any time without any direct impact on the user because the interface between the abstract data type and its users remains the same. Of course, the user should see the impact of changing a slow implementation by a faster one, or even a buggy implementation by a correct one! What is meant here is that no changes should be necessary to his/her code to use the new implementation.

2.1.1 Stacks

The stack abstract data type stores objects in a last-in, first-out basis, i.e. the last object stored will be the first one to be retrieved. It supports basically only two operations:

- 1) Push: an object is stored in the stack. This object becomes the top of the stack, and the former top of stack is moved (conceptually) one place down.
- 2) Pop: an object is retrieved from the stack, and the object below the top object becomes the new top of the stack.

Stacks are implemented usually using an array or a list. Both arrays and list containers support the basic stack operations. We will discuss stacks more later in this chapter.

2.1.2 Strings

Strings are a fundamental data type for a wide type of applications. We have seen in the preceding chapters the problems that C strings have. Here is a proposal for a string library that allows you to avoid those problems.

The whole source code is available to you for easy customization. At any time you can build a specialized version of it, to fit a special need or to change one of the design parameters.

The string library is based in the abstract notion of a container, i.e. a common set of interface functions, that allow you some latitude when building your program. Using the abstract indexing notation, you can change your data representation from a vector to an array, or even from a list to a vector without being forced to change a lot in your own programs.

Strings are a sequence of characters. We can use one or two bytes for storing characters. Occidental languages can use fewer characters than Oriental languages, that need two

bytes to store extended alphabets. But occidentals too, need more than one byte in many applications where mathematical symbols, drawings and icons increase the size of the available alphabet beyond the 256 limit of a byte.

We classify then strings in multi-byte sequences of characters, (alphabet is smaller than 256) or wide character strings, where the alphabet can go up to 65535 characters. The library handles both character types as equivalent.

The definition of a string looks like this:

```
typedef struct _String {
    size_t count;
    unsigned char *data; // wchar_t for wide strings
    size_t capacity;
} String;
```

The count field indicates how many characters are stored in the String. Follows the data pointer, the allocated size, and the type of the string: either one or two bytes characters.

We store a pointer to the data instead of storing the data right behind the descriptor. We waste then, `sizeof(void *)` for each string but this buys an increased flexibility. Strings are now resizeable since we can change the pointer and reallocate it with a different size without having to allocate a new string. This is important since if there are any pointers to this string they will still be valid after the resize operation, what would not be the case if we had stored the characters themselves in the data structure.

Text is stored either coded in one byte characters, or in UNICODE (multi-byte) numbers of two bytes. There are then two kinds of Strings: a single byte representation called StringA, and a multi-byte representation called StringW. The type “String” can be assigned to the first or the second according to the value of a defined constant UNICODE.

The only difference between the two types of string is the type of the data the “data” pointer points to: in one case is a `char *`, in the other is a `wchar_t`.

```
#ifdef UNICODE
#define String StringW
#else
#define String StringA
#endif
```

Of course if you do not use UNICODE, you can still use wide character strings by explicitly naming them StringW.

The advantage of this schema is that there is only one set of names and a simple rule for using all the string functions.

The functions are patterned after the traditional C functions, but several functions are added to find sets of characters in a string, or to read a file into a string.

In general, the library tries as hard as possible to mimic the known functions and notation of the existing C strings.

The string library introduces the concept of iterators, or fat pointers. This data structure consists of a pointer to some character within the container, in this case the string, a count, and a pointer to the whole strings. String pointers are called Stringp, and they come in two flavors too: wide and single byte strings.

There are two properties of the string you will want to have access to: its length, and its capacity. The length is the number of characters that the string contains, and the capacity is the number of characters the string can store before a resize operation is needed. You can access those properties with

```
String s1;
...
int len = get_size(s1);
```

2.2 Design criteria

What are the design decisions that make the base of the library?

2.2.1 Memory management

The first versions of the library contained for each container a pair of pointers for the memory allocation and memory release functions. This increases the size of the library, and considerably increases the number of things that can go wrong.

Each container had its own memory management, for releasing elements and maintaining free lists. This was cumbersome and has been discarded. Memory management is better done by the memory manager, i.e. the garbage collector.

Since lcc-win32 features a garbage collector (GC) in the standard distribution, this simplifies the library. Other environments like Java or the recent .Net architecture of Microsoft feature as a standard library feature the garbage collector too.

2.2.2 The handling of exceptions

All containers check any indexed access for errors. Badly formed input is detected at run time and the program (by default) is stopped with an error message. The interface uses the 'require' macro, exactly as defined in the Safer C library. This macro is the following:

```
#define require(constraint) \
    ((constraint) ? 1 : ConstraintFailed(__func__, #constraint, NULL) )
```

The function ConstraintFailed calls the user defined handler function. That function can return (or not) depending what do you want to do with the exception. By default the handler function is set to show the error message and exit the program.

You can change the default function that is called with the helper functions

```
constraint_handler_t set_constraint_handler_s(constraint_handler_t
                                              handler);
constraint_handler_t get_constraint_handler_s(void);
```

The type constraint_handler_t is defined in stdlib.h as follows:

```
typedef void (*constraint_handler_t)( const char * restrict msg,
                                     void * restrict ptr, errno_t error);
```

2.2.3 Efficiency considerations

The C language has an almost obsessive centering in “efficiency”. Actually, as everybody knows, efficiency depends on the algorithm much more than in the machine efficiency with which operations are coded. Length delimited strings are by nature more efficient than normal C strings since the often used `strlen` operation takes just a memory read, instead of starting an unbounded pointer memory scan seeking the trailing zero.

Efficiency must be weighted against security too, and if we have to choose, when implementing the container library, security has been always more important than machine efficiency. Some operations like array bound checking can add a small overhead to the run time, but this will be of no concern to the vast majority of the applications done with this library. Using a hash table will be always more efficient than a plain linear scan through a hastily constructed table. Even if we code the table lookup in assembler.

In this version, efficiency has been left out. The weight of the effort has gone into making a library that works and has fewer bugs.

2.2.4 C and C++

If you know the C++ language, most of these things will sound familiar. But beware. This is not C++. There is no “object oriented” framework here. No classes, instantiated template traits, default template arguments, default copy constructors being called when trying to copy a memory block. Here, a memory block copy is that: a memory block copy, and no copy constructors are called anywhere. Actually, there are no constructors at all, unless you call them explicitly.

You can do copy constructors of course, and the library provides `copy_list`, `copy_bitstring`, etc. But you call them explicitly, the compiler will not try to figure out when they should be called. All this makes programs clearer: we know what is being called, and the calls do not depend any more on the specific compiler version.

Yes, the compiler does some automatic calls for you. You can redefine operators, and you have generic functions. This introduces many problems you are aware of, if you know C++. Here however, the complexity is enormously reduced, since there are no systematic implicit calls, and objects are not automatically clustered in classes. You can make classes of objects of course, but you do it explicitly. No implicit conversions are granted in the language itself.

Here we have a library for using simple data structures. Nothing is hidden from view behind an opaque complex construct. You have the source and a customizing tool.

C has a formidable power of expression because it gives an accurate view of the machine, a view that has been quite successful.

C++ is a derived language from C. It introduced the notion of well classified “objects”, an idea that was put forward by many people, among others Bertrand Meyer, and Xerox, with their Smalltalk system.

Unfortunately, what was a good paradigm in some situations becomes a nightmare in others. When we try to make a single paradigm a “fits all” solution, the mythical silver bullet,

nothing comprehensible comes out. To be “object oriented” meant many things at once and after a while, nothing at all.

For C++ the object oriented framework meant that functions were to be called automatically by the compiler, for creating and destroying objects. The compiler was supposed to figure out all necessary calls. This is one solution for the memory management problem.

Another solution, and the one lcc-win32 proposes, is to use a garbage collector. C++ did not introduce a GC for reasons I have never understood. This solution is much simpler than making a complex compiler that figures out everything automatically. The destructor is the GC that takes care of the leftovers of computation.

Then, it becomes possible to make temporary objects without worrying about who disposes of temps. The GC does. Operators can return dynamically allocated temporary objects without any problem. The equivalent of C++ automatic object destruction is attained with the GC, and the complexity of the software is reduced.

At the same time, bounds checked arrays and strings become possible. A general way of using arrays and other data structures is possible.

2.3 Implementation

2.3.1 Creating strings

The simplest way to create strings is to assign them a C string. For instance:

```
String s = "This is a string";
```

The compiler will generate here a call to the overloaded assignment operator for Strings, that will construct a String from a zero terminated string, i.e. a `char *`. A problem is, of course, what to do at the global level since no function call can be generated. In the existing implementation it is not possible to associate a String to a immediate character string at the global level.

To allocate a string that will contain at least `n` characters you write:

```
String s = new_string(n);
```

The primitive `new_string` is a versatile function. It can accept also a character string:

```
String s = new_string("This is a string");
```

or a double byte character string:

```
String s new_string(L"This is a string");
```

2.3.2 Copying

When you assign a String to another, you make a *shallow* copy. Only the fields of the String structure will be copied, not the contents of the string. To copy the contents of the string you use the `copy` function or its synonym `Strdup`:

```
String s2 = Strdup(s1);
```

This `Strdup` does *not* call `malloc`, and you are not supposed to free the allocated memory. You can provide a starting index to `Strdup`, and it will copy from that character position to the end of the string.

```
String s2 = strdup(s1,2); // copy starting with the third character
```

Remember that index starts with zero in C. You can also provide an end index, meaning `Strdup` will duplicate characters beginning with the start index and ending with the end index (inclusive). For instance:

```
String s1 = "This is a character string";
```

Destructively copying a string into another is done with the `Strcpy` function.

```
String s1 = "a", s2 = "abcd";  
Strcpy(s1,s2); // Now s1 contains "abcd"
```

To copy a certain number of characters into another string, you use the `Strncpy` function:

```
String s1 = "abc", s2 = "123456789";  
Strncpy(s1,s2,5); // Now s1 contains "12345";
```

Note that this `Strncpy` always builds correct strings, i.e. there is no inconsistencies in the result.

2.3.3 Accessing the characters in a String

The `[]` notation is used to access the characters as in normal C character strings. Given the string:

```
String s1 = "abc";
```

you access each character with an integer expression:

```
int c = s1[1]; // Now c contains 'b'
```

You assign a character in the string with:

```
s1[0] = 'A';
```

Now the string contains "Abc".

Note that mathematical operations with characters are not supported. You can't write:

```
s1[0] += 2; // Wrong
```

The rationale behind this decision is that characters are characters and not numbers. You can of course do it if you do:

```
int c = s1[0];  
c += 2;  
s[0] = c;
```

2.3.4 Comparing strings

You can compare two strings with the `==` sign, or with the `Strcmp` function. `Strcmp` returns the lexicographical order (alphabet order) for the given strings. The equals sign just compares if the strings are equal. Both types of comparison are case sensitive. To use a case

insensitive comparison use `Strcmpi`. Note that this implementation doesn't support the case insensitive wide char comparison yet.

2.3.5 Relational operators

The relational operators can be defined like this:

```
int operator ==(const String & string1,
               const String & string2)
{
    if (isNullString(string2) && isNullString(string1))
        return 1;
    if (isNullString(string2) || isNullString(string1))
        return 0;
    struct Exception *exc = GetAnExceptionStructure();
    PRECONDITION((Strvalid(string1) && Strvalid(string2)), exc);
    if (string1.count != string2.count)
        return 0;
    return !memcmp(string1.content, string2.content, string1.count);
}
```

We check for empty strings, that can be compared for equality without provoking any errors. Two empty strings are considered equal. If either of the strings is empty and the other isn't, then they can't be equal.

Those tests done, both strings must be valid. They are equal if their count and contents are equal. Note that we use `memcmp` and not `strcmp` since we support strings with embedded zeroes in them.

The wide character version differs from this one only in the length of the memory comparison.

The function “`isNullString`” tests for the empty string, i.e. a string with a count of zero, and contents `NULL`.

An empty string is returned by some functions of the library to indicate failure. It is semantically the same as the `NULL` pointer.

The other relational operators are a bit more difficult. Here is “less” for instance:

```
int operator < (const String & s1, const String & s2)
{
    bool s1null = isNullString(s1);
    bool s2null = isNullString(s2);
    if (s1null && s2null)
        return 0;
    if (s1null && !s2null)
        return 1;
    if (!s1null && s2null)
        return 0;
    struct Exception *exc = GetAnExceptionStructure();
    PRECONDITION((Strvalid(s1) && Strvalid(s2)), exc);
    if (s1.count == 0 && s2.count != 0)
        return 1;
    if (s1.count && s2.count == 0)
        return 0;
}
```

```

    int len = s1.count < s2.count ?
        s1.count : s2.count;
    return memcmp(s1.content, s2.content, len) < 0;
}

```

We have to differentiate between a NULL string, an empty string and a valid or invalid string. This is the same as in standard C where we differentiate between NULL and "", the empty string. A pointer can have a NULL value, a valid value pointing to an empty string, or an invalid value pointing to nowhere.

We put NULL and empty strings at the start of the lexicographical order, so they are always less than non empty strings. Note that we compare only as much as the shorter of both strings. This is important, because we wouldn't want that memcmp continues comparing after the end of the shorter string. Since we have the length of the string at hand, the operation is very cheap, a few machine instructions.

2.3.6 Dereferencing strings

Another operator worth mentioning is the pointer dereference operator '*'.

In C, a table is equivalent to a pointer to the first element. With the definitions:

```

char str[23];
*str = 0;

```

The expression *str = 0 is equivalent to str[0] = 0. We can mimic this operation with the redefinition of the '*' operator:

```

char * operator *(StringA &str)
{
    require(StrvalidA(str) && str.count > 0);
    return &str.content[0];
}

```

This operator must always return a pointer type. In this case it returns a pointer to the first character. This allows to support the *str = 'a' syntax. There are several differences though:

- 3 An empty string cannot be accessed. In traditional C it is possible to write *s = 'a', even if "s" is an empty string. This destroys the string, of course, it is no longer zero terminated, but no trap occurs at the point of the assignment since "s" points to valid memory. Traps occur later, when a non zero terminated string destroys the whole program. Here, any access to an empty string will provoke an exception.
- 4 The syntax *s++ is not supported. You can't increment a string structure. You can only increment a pointer to a string (a Stringp). Both types are distinct, to the contrary of traditional C where they are both the same.
- 5 You may wonder when you see that this operator returns a naked char pointer. Wouldn't this mean that this pointer could be misused in the code using the library? Happily for us, the compiler dereferences immediately the result of this operator, so the pointer can't be used anywhere else, or even be assigned to something. If you try to

write “char *p = *s” it will provoke a compile time error since you are assigning a char to a pointer to char, what is not allowed.

Is it necessary to do this?

We could have decided that the user will always use the array notation (the []), but in general, it is better to make the string package behave as much as possible (but not more) as traditional C strings. The objective is that people do not have to retrain themselves to use this package. As much as possible from the old syntax should be understood.

2.3.7 Imitating pointer addition

With traditional C strings it is valid to add an integer to a string pointer to obtain a pointer to the middle of the string. We can mimic this behavior by overloading the addition operator.

```
StringpA operator+(StringA &s1,int n)
{
    StringpA result;
    require(StrvalidA(s1) &&
           n >= 0 &&
           n < s1.count);
    result.count = s1.count - n;
    result.content = s1.content + n;
    result.parent = &s1;
    return result;
}
```

We test for validity of the given string and check that the offset passed is correct. We return a Stringp (not a String!) that is initialized to point to the specified offset. The result of this operation is to produce a fat pointer and not a String. This will cause problems, since even if the library tries to hide most differences, a Stringp is another kind of beast than a normal String. Note too that adding negative offsets is no longer possible.

2.3.8 String pointer operations

The library introduces the notion of string pointer structures, i.e. “fat” pointers that contain, besides the normal pointer to the contents, a count and a pointer to the parent string. Pointer operations on this structures are few, and they try to mimic the normal pointer behavior.

This pointers can be used as iterator objects to go through a portion or all the string contents.

You can obtain an iterator to the beginning of the string with:

```
String str = new_string(15);
Stringp pStr = begin(str);
```

The polymorphic function “begin” returns a pointer to the beginning of a sequential container.

The operations supported with string pointers are:

2.3.9 Pointer subtraction

```
int operator -(Stringp &s1, Stringp &s2)
{
    require(Strvalidp(s1) && Strvalidp(s2) &&
           s1.parent == s2.parent);
    return s1.content - s2.content;
}
```

The operator verifies that both pointers are valid, and that they point to the same string. The operation is more restricted than in traditional C since a subtraction operation is considered invalid if the pointers do not point to the same parent.

2.3.10 Addition of pointer and integer

This operation moves the given pointer forwards or backwards a specified number of characters.

```
Stringp operator+=(Stringp &s1, int n)
{
    require(Strvalidp(s1) && n < s1.count);
    s1.count -= n;
    s1.content += n;
    return s1;
}
```

2.3.11 Comparisons of a string pointer with zero

Several string functions return an invalid string pointer to indicate failure. It is practical that this pointer equals to NULL, so that code snippets like:

```
if (!Strchr(String, '\n')) {
}
```

work as intended. We overload the operator `!=` and the operator `==` to mimic this behavior:

```
int operator != (const Stringp & string1, int i)
{
    if (isNullStringp(string1) && i == 0)
        return 0;
    return 1;
}
```

We allow only comparisons with zero. The operator `==` is very similar:

```
int operator == (const Stringp & string1, int i)
{
    if (i == 0 && isNullStringpA(string1))
        return 1;
    return 0;
}
```

2.4 String functions

The string library supports all the standard functions defined in `string.h`. The names chosen are the same, with the first letter in upper case: `strcat` is `Strcat`, `strcmp` is `Strcmp`, etc. Most of those functions are very simple, the specifications for the C run time library are quite primitive.

2.4.1 Comparing strings

Here is, for instance, the `Strcmp` function:

```
int overloaded Strcmp(String & s1, String & s2)
{
    require(Strvalid(s1) && Strvalid(s2));

    /* conversion widening the smaller to the wider string type */
    if (0 == s1.count && s2.count == 0){
        return 0;
    }
    if (0 == s1.count){
        return -1;
    }
    if (0 == s2.count){
        return 1;
    }
    int len = s1.count < s2.count ? s1.count : s2.count;
    return memcmp(s1.content, s2.content, len);
}
```

`Strcmp` accepts also a `char *`, so that users can write:

```
if (!Strcmp(str, "Annie")) { ... }
```

This syntax is widespread, and it is important to support it. Here is a different version of `Strcmp` that accepts `char` pointers.

```
int overloaded Strcmp(String & string, const char* str)
{
    int len;
    require(Strvalid(string) && str != NULL);
    len = strlen(str);
    if (0 == string.count && len == 0) return 0;
    if (0 == string.count) return -1;
    if (0 == len) return 1;
    len = len < string.count ? len, : string.count;
    return memcmp(string.content, str, len);
}
```

There are other variations, for supporting wide chars, and changing the order of the arguments.

2.4.2 Joining strings

To join several C strings into a single string the library proposes `Strcatv`¹. It receives a series of strings, transforming them into a single `String`.

```
String Strcatv(char *s,...)
{
    int len = strlen(s);
    va_list ap,save;
    char *next,*p;
    StringA result;

    va_start(ap,s);
    save = ap;
    next = va_arg(ap,char *);
    while (next) {
        len += strlen(next);
        next = va_arg(ap,char *);
    }
    va_start(ap,s);
    result = new_stringA(len);
    strcat(result.content,s);
    p = result.content + strlen(s);
    next = va_arg(ap,char *);
    while (next) {
        while (*next) {
            *p++ = *next++;
        }
        next = va_arg(ap,char *);
    }
    *p = 0;
    result.count = p - result.content;
    return result;
}
```

We make two passes over the strings, first collecting their lengths, then joining them. We avoid using `strcat` when joining the strings, since that would be very inefficient. The standard library `strcat` needs to find the terminating zero, what is more and more expensive as the length of the result string grows. Instead, we use a roving pointer that adds right at the end of the previous string the new one.

2.4.3 Accessing strings

The general accessing function for `Strings` is:

```
int operator[ ](String s,size_t index);
```

This operator returns the character at the given position. The operator checks that `s` is a valid string, and that the index is less than the length of the string.

```
int operator[](const String& string, size_t index)
```

1. This is essentially the same function as `Str_catv` from Dave Hanson's C Interfaces and implementations.

```

{
    require(Strvalid(string) && index < string.count);
    return string.content[index];
}

```

Single byte versions will use `StrvalidA` and receive `StringA` strings, double byte versions will use `StringW` and call `StrvalidW`. We will use the generic term `String` when we mean either `StringW` or `StringA`.

The operation indexed assignment (operator `[] =`) is handled by:¹

```

int operator[]=(String & string, size_t index, int new_val)
{
    require(Strvalid(string) && index < string.count);

    string.content[index] = new_val;
    return new_val;
}

```

2.4.4 Adding and inserting characters

Inserting a single character is accomplished by the `Strinsert` function, or its alias “insert” that should work with all containers.

```

bool Strinsert(String &s, size_t pos, int newval)
{
    char *content;
    int new_capacity;
    require(Strvalid(s));
    require(pos < s.count);

    new_capacity =
        calculate_new_capacity(s.capacity, s.capacity+1);
    if (new_capacity > s.capacity) {
        content = allocate_proper_space(new_capacity, SMALL);
        if (content == NULL)
            return true;
        memcpy(content, s.content, s.count);
        s.capacity = new_capacity;
        s.content = content;
    }
    else content = s.content;
    memmove(content+pos+1, content+pos, s.count-pos);
    s.count++;
    content[pos] = newval;
    content[s.count] = 0;
    return true;
}

```

The preconditions are a valid string and a valid index. We call the “`calculate_new_capacity`” function to get an estimate of the best new size if a string resize

1. Note that in C++ is not possible to distinguish between this two. There is no `[] =` operator.

is needed. This is a relatively expensive operation, so we always allocate more space than strictly needed, to avoid resizing the string at each character added.

After we have ensured that we have the space needed, we make place for the new character by moving the tail of the string one position up.

The erase function `Strdelete` will remove a character from the string at the indicated position.

```
bool Strdelete(String &s, size_t pos)
{
    int element_size;
    char *pcontents;
    wchar_t *pwcontents;
    if (!Strvalid(s))
        return false;
    if (s.count == 0)
        return false;
    require(pos < s.count);
    pcontents = s.content;
    if (pos < s.count-1) {
        memmove(pcontents+pos, pcontents+pos+1, s.count-pos-1);
    }
    s.count--;
    pcontents[s.count] = 0;
    return true;
}
```

We allow erasing characters from an empty string, and just return false if that is the case. This allows for loops that will use the result of `Strdelete` to stop the iteration. In this case, we just return false. Otherwise we require a valid `String` and a valid index.

A small optimization is performed when the character to erase is the last character in the string. We spare us a `memmove` call by just setting the character to zero and decreasing the count field of the string structure. Otherwise, we have to move the characters from the position we are going to use one position down.

2.4.5 Mapping and filtering

A mapping operation in a string means applying a function to each character in the source string, and storing the result of that call in the new string that is the result of the operation.

```
String Strmap(String & from, int (*map_fun) (int))
{
    String result;

    if (map_fun == NULL)
        return Strdup(from);
    require(Strvalid(from) && map_fun != NULL);
    result = new_string(from.count);
    result.count = from.count;
    for (int i = 0; i < from.count; ++i) {
        result.content[i] = map_fun(from.content[i]);
    }
    return result;
}
```

```
}
```

Note that a NULL function pointer means that the identity function¹ is assumed, and the whole operation becomes just a copy of the source string. We do not need to test for the validity of the source string in that case since `Strdup` does that for us.

We allocate a string that will contain at most so many characters as the source string including always a terminating zero. If the predicate function filters most of the characters, the string will be almost empty. Since we keep track of this in the capacity field of the `String` structure, this is less terrible than it looks like. The other solution would be to call twice the predicate function, but that would be very expensive in CPU usage.

A similar function is `Strfilter`, that will output into the result string only those characters that satisfy a boolean predicate function. `Strfilter` will accept either `Strings` or character strings. Here is the version that uses character strings:

```
String overloaded Strfilter(String & from, char *set)
{
    String result;
    int setlength;

    require(Strvalid(from) && set != NULL);
    setlength = strlen(set);
    result.content = allocate_proper_space(1+from.count, SMALL);
    if (result.content == NULL) {
        memset(&result, 0, sizeof(result));
        return result;
    }
    result.capacity = 1+from.count;
    int j = 0;
    for(int i = 0; i < from.count; ++i) {
        for (int k=0; k<setlength;k++) {
            if (from.content[i] == set[k]) {
                result.content[j++] = set[k];
                break;
            }
        }
    }
    result.count = j;
    return result;
}
```

We need a valid string and a non-null set. We allocate space for the string including the terminating zero with the “`allocate_proper_space`” function. That function will raise an exception if no more memory is left and terminate the program. In case the user has overridden that behavior, we return an invalid `String`.

This is the single byte version, and we indicate this to “`allocate_proper_space`” with the `SMALL` parameter. For the wide character set we would replace that with `WIDE`.

1. The identity function in C is:
`int identity(int c) { return c; }`

If the allocation succeeds we set the capacity field, and we select the characters to be included in the result. At the end, we set the count field to the number of chars found that matched the predicate.

2.4.6 Conversions

To interact easily with other software we need to convert Strings in traditional strings, and we have to allow for converting traditional strings into the String structure. We use the overloaded cast operator to give the conversions the traditional meaning in C.

```
StringA operator()(char* cstr)
{
    StringA result;
    require(cstr != NULL);
    result.count = strlen(cstr);
    result.capacity = calculate_new_capacity(0,1+result.count);
    result.content = allocate_proper_space(result.capacity,SMALL);
    if (result.content == NULL) {
        return invalid_stringA;
    }
    strcpy(result.content,cstr);
    return result;
}
```

We build a new String structure from scratch. We require a valid C string, and calculate its length. We determine the best capacity for the new string, allocate the contents and copy.

We use this operator with a cast, for instance:

```
void printName(String &s);
...
printName((String)"Annie");
```

The inverse operator is the (much simpler) cast from a String to a char*.

```
char *operator()(StringA &str)
{
    if (isNullStringA(str))
        return NULL;
    str.content[str.count] = 0;
    return str.content;
}
```

We use this operator like this:

```
String str;
printf("%s\n", (char *)str);
```

Conversions from string pointers to char pointers is more difficult if we want to support pointers that span only a subset of the string. For instance if we have a pointer of length 2 that points to the third character of the string “This is it”, the pointer points to the first ‘i’ and includes the ‘i’ and the ‘s’ but not more. In that case we need to copy the contents before passing them to the calling function.

```
char *operator()(Stringp &strp)
{
    char *result;
```

```

    result = strp.content;
    if (result[strp.count] == 0)
        return result;
    result = allocate_proper_space(strp.count+1,SMALL);
    if (result == NULL)
        return NULL;
    memcpy(result,strp.content,strp.count);
    return result;
}

```

2.4.7 File operations

Files can be read as a whole into a string for later processing. This is a similar operation as building a memory mapped file.

```

StringA overloaded Strfromfile (char * file_name,int binarymode)
{
    FILE *fp;
    StringA result;

    if (binarymode)
        fp = fopen(file_name, "rb");
    else
        fp = fopen(file_name, "r");
    if (NULL == fp){
        return invalid_stringA;
    }
    size_t needed = 0;
    int i_rval = 0;
    i_rval = fseek(fp, 0, SEEK_END);
    if (0 != i_rval) {
        fclose(fp);
        return invalid_stringA;
    }
    needed = ftell(fp);
    i_rval = fseek(fp, 0, SEEK_SET);
    if (0 != i_rval){
        fclose(fp);
        return invalid_stringA;
    }
    result.content = allocate_proper_space(needed+1,SMALL);
    if (result.content == NULL)
        return invalid_stringA;
    result.capacity = needed+1;
    size_t u_rval = fread(result.content, 1, needed, fp);
    fclose(fp);
    if (u_rval != needed){
        return invalid_stringA;
    }
    result.count = needed;
    return result;
}

```

According to the “binary mode” parameter, we read the file including the `\r\n` sequence into the contents or not. We determine the file size, then we read the string into the string contents.

Another often needed operation is reading a line from a string.

```
int Strfgets(StringA & source, int n, FILE *from)
{
    int i,val;
    if (StrvalidA(source) == 0 || n > source.capacity) {
        source.content = allocate_proper_space(n+1, SMALL);
        if (source.content == NULL)
            return 0;
        source.capacity = n+1;
    }

    for (i = 0; i < n; ) {
        val = fgetc(from);
        if (val == EOF || val == '\n')
            break;
        if (val != '\r') {
            source.content[i++] = val;
        }
    }
    source.content[i] = 0;
    source.count = i;
    return (i);
}
```

2.4.8 Reversing a String

Reversing the order of elements in a string is accomplished by our `Strreverse` function.

```
bool Strreverse(String &s)
{
    if (!Strvalid(s))
        return false;
    char *p = s.content;
    char *q = p;
    int i = s.count;

    p += i;
    while (i-- > 0) {
        *q++ = *--p;
    }
    *q = 0;
    return true;
}
```

2.4.9 Searching text

The `Strstr` searches a pattern in a given string, and returns a fat pointer to a string (`Stringp`), or an invalid string if not found.

```

Stringp overloaded Strstr(String & string, String & find_this)
{
    char *strp,*end;
    int first_char,r;

    require(Strvalid(string) && Strvalid(find_this));
    first_char = find_this.content[0];
    strp = memchr(string.content,first_char,string.count);
    if (strp) {
        end = string.content + string.count;
        while (strp < end) {
            if (!memcmp(strp,find_this.content,find_this.count))
                return new_stringp(string,strp);
            else
                strp++;
        }
    }
    return invalid_stringp;
}

```

We search the first character of the string to search in the string. Each time we find it, we test if the search pattern is there. If yes we return a new string pointer, otherwise we go on until we reach the end of the string.

To search a character in a string we use the Strchr function:

```

Stringp overloaded Strchr(String & string, int element)
{
    struct Exception *exc = GetAnExceptionStructure();
    PRECONDITION((Strvalid(string)), exc);
    char *p = memchr(string.content,element,string.count);
    if (p){ // found
        return new_stringp(string,p);
    }
    return invalid_stringp;
}

```

2.4.10 Making a string from a pipe

The function Strfrompipe will start a program and capture all its textual output into a String. This is useful for many GUI applications that want to show the results of a program in a different way, and many others.

The algorithm used is very simple: we start the command indicated in the string argument with the process using a redirected standard output into a temporary file. We read then this file into a string a return the result.

```

StringA overloaded Strfrompipe(String &Cmdline)
{
    STARTUPINFO startInfo;
    char *p;
    int processStarted,m;
    String result = invalidStringA;
    HANDLE hWritePipe;
    LPSECURITY_ATTRIBUTES lpSa=NULL;

```

```

SECURITY_ATTRIBUTES sa;
SECURITY_DESCRIPTOR sd;
ThreadParams tparams;
DWORD Status;
PROCESS_INFORMATION pi;
char *tmpfile = tmpnam(NULL);
char cmdline[1024];
char arguments[8192];

memset(cmdline, 0, sizeof(cmdline));
memset(arguments, 0, sizeof(arguments));
    // Check for quoted file names with spaces in them.
if (Cmdline.content[0] == '"') {
    p = strchr(Cmdline.content+1, '"');
    if (p) {
        p++;
        strncpy(cmdline, Cmdline.content, p-Cmdline.content);
        cmdline[p-Cmdline.content] = 0;
        strncpy(arguments, p, sizeof(arguments)-1);
    }
    else return invalid_stringA;
}
else {
    strncpy(cmdline, Cmdline.content, sizeof(cmdline));
    p = strchr(cmdline, ' ');
    if (p) {
        *p++ = 0;
        strncpy(arguments, Cmdline.content);
    }
    else arguments[0] = 0;
}
if (IsWindowsNT()) {
    InitializeSecurityDescriptor(&sd,
        SECURITY_DESCRIPTOR_REVISION);
    SetSecurityDescriptorDacl(&sd, TRUE, NULL, FALSE);
    sa.nLength = sizeof(SECURITY_ATTRIBUTES);
    sa.bInheritHandle = TRUE;
    sa.lpSecurityDescriptor = &sd;
    lpsa = &sa;
}
memset(&startInfo, 0, sizeof(STARTUPINFO));
startInfo.cb = sizeof(STARTUPINFO);
tparams.file = fopen(tmpfile, "wb");
if (tparams.file == NULL) {
    return invalid_stringA;
}
hWritePipe = (HANDLE)_get_osfhandle(_fileno(tparams.file));
startInfo.dwFlags = STARTF_USESTDHANDLES|STARTF_USESHOWWINDOW;
startInfo.wShowWindow = SW_HIDE;
startInfo.hStdOutput = hWritePipe;
startInfo.hStdError = hWritePipe;
result = invalid_stringA;
processStarted = CreateProcess(cmdline, arguments, lpsa, lpsa, 1,
    CREATE_NEW_PROCESS_GROUP|NORMAL_PRIORITY_CLASS,

```

```
        NULL, NULL, &startInfo, &pi);
if (processStarted) {
    WaitForSingleObject(pi.hProcess, INFINITE);
    GetExitCodeProcess(pi.hProcess, (unsigned long *)&Status);
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
    m = 1;
}
else m = 0;
if (tparams.file)
    fclose(tparams.file);
if (m) {
    result = Strfromfile(tmpfile, 1);
}
remove(tmpfile);
return(result);
}
```

2.4.11 Searching strings

There are several find functions. Here is an overview:

This functions perform primitive string searches. For more sophisticated searches use the

<i>Function</i>	<i>Description</i>
<code>Strfind_first_of</code>	Finds the first character in a string that matches a given set. For instance using the set of the tab character and space, it finds the first whitespace character in a string.
<code>Strfind_last_of</code>	Finds the last character that matches the given set. Using the example above it would find the last white space character.
<code>Strfind_first_not_of</code>	Find the first char not in the given set.
<code>Strfind_last_not_of</code>	Find the last char not in the given set
<code>Strcspn</code>	Finds the index of the first char that matches any of the given set. The difference with <code>Strfind_first_of</code> is in the return value. <code>Strcspn</code> returns the length of the string in case of error, <code>Strfind_first_of</code> returns -1.
<code>Strchr</code>	Finds the first occurrence of a given character in a string.
<code>Strrchr</code>	Finds the last occurrence of a given character in a string.
<code>Strstr</code>	Finds the first occurrence of a pattern in a string.
<code>Strspn</code>	Finds the index of first char that doesn't match a character in a given set. The difference with <code>Strfind_first_not_of</code> is in the return value: <code>Strspn</code> returns the string length when no match is found instead of -1.

regular expression package or the perl regular expression package (pcre).

Finding the first whitespace character in a string:

```
Strfind_first_of(Source, "\t ");
```

Finding the end of the first word in a string:

```
char *alphabet =
"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz";
Strfind_first_not_of(Source, alphabet);
```

Finding a word in a string:

```
String s = "A long short sentence";
Stringp ps = Strstr(s, "tence"); // Now ps points to "tence"
```

Note that `Strstr` returns a pointer structure, not a new string. You can convert the pointer into a string with the `Strdup` function.

2.4.12 `Strfind_first_of`

```
int overloaded Strfind_first_ofA(StringA &s, StringA &set)
{
```

```

    require(StrvalidA(s) && StrvalidA(set));
    char *strcontents = s.content,*p;
    int i,j,count = s.count,setcount = set.count;
    int c;

    if (set.count == 0 || s.count == 0)
        return -1;
    for (i=0; i<count;i++) {
        p = set.content;
        c = strcontents[i];
        for (j=0; j<setcount;j++) {
            if (c == p[j])
                return i;
        }
    }
    return -1;
}

```

2.4.13 Joining strings

To join a string to another there are several functions described below. Note that the operator '+' is not used for joining strings. The reasoning behind this is that the addition of strings in the sense used here is non-commutative. "abc" + "dce" give "abcdce" but "dce"+"abc" gives "dceabc", what is quite different. An overloading of the + operation is not warranted. Strings are not numbers, and if we did this overloading, we could come to the interesting conclusion that "1"+"1" = "11"...

<i>Function</i>	<i>Description</i>
Strcat	Joins two strings, changing the first.
Strncat	Joins up to n characters from the second argument to the first.
Strchcat	Inserts a character at the end of the string.
Strcatv	Joins several C strings into a single String.

Given the string

```
S1 = "abcde";
```

The call

```
Strcat(S1, "fgh");
```

will modify the contents of S1, that will after the call contain "abcdefgh". It is possible that no resize operation is necessary, if the capacity of S1 was already big enough to accommodate both strings. Otherwise the library resizes S1.

Resizing operations are expensive when done frequently. To avoid them, it is better to resize the string before doing the joining in such a way that the resize operation is done once, instead of several times.

2.4.14 Strncat

```
bool overloaded Strncat(StringA & to, StringA & from, size_t
elements_to_cat)
{
    if (!StrvalidA(to)){
        if (StrvalidA(from)) {
            to.count = from.count;
            to.capacity = from.capacity;
            to.content = GC_malloc(from.capacity);
            memcpy(to.content, from.content, from.count);
            return true;
        }
        return false;
    }
    require(StrvalidA(from), exc);
    size_t needed_space = elements_to_cat + to.count+1;
    to = str_newA(needed_space, to);
    strncat(to.content, from.content, elements_to_cat);
    to.count = needed_space-1;
    to.content[to.count] = 0;
    return true;
}
```

2.4.15 Strcat

This just builds a string from the “from” argument and calls Strncat.

```
bool overloaded Strcat(StringA &to, char* from)
{
    StringA s;

    if (from == NULL)
        return true;
    s.count = strlen(from);
    s.capacity = s.count+1;
    s.content = from;
    return Strncat(to, s, s.count);
}
```

2.4.16 Strmap

This function applies a function in sequence to all characters in a string. It is useful for implementing filters or other transformations to a given string. For instance if we want to change all characters of a string into upper case, we can do the following:

```
#include <strings.h>
#include <ctype.h>
int change(int c)
{
    if (islower(c))
        c = toupper(c);
    return c;
}
```

```

}

int main(int argc, char *argv[])
{
    String s, s1;
    for (int i=0; i<argc; i++) {
        s = argv[i];
        s1 = Strmap(s, change);
        printf("%s\n", (char *)s1);
    }
    return 0;
}

```

This program will output its arguments transformed in uppercase. This is an example, of course. There is already a function that does that: `Strcmpi`.

2.4.17 Strtrim

This function eliminates superfluous blanks and tabs from the given character string.

```

#include <ctype.h>
#include "str.h"
int Strtrim(StringA &str)
{
    char *src, *dst, *start;

    if (!StrvalidA(str)) return -1;
    src = str.content, dst = src, start = src;

    while (isspace(*src))
        src++;
    do {
        while (*src && !isspace(*src))
            *dst++ = *src++;
        if (*src) {
            *dst++ = *src++; // Keep one space
            while (isspace(*src) && *src != '\n' && *src != '\r')
                src++;
        }
    }
    while (*src);
    if (dst != start && isspace(dst[-1]) && dst[-1] != '\n')
        dst--;
    *dst = 0;
    str.count = dst - start;
    return 0;
}

```

2.4.18 Filters

The `Strfilter` function builds a new string with the characters that fulfill a predicate function passed to it as an argument or is a member of a given set. For instance, if you want to extract all numeric data from a string that contains numbers and other data you use:

```

bool isnumeric(int c)
{
    return c >= '0' && c <= '9';
}
String s1 = "Subject age is 45 years";
String s2 = Strfilter(s1,isnumeric); // Now s2 is "45"

```

Strfilter will also accept a String or char/wide character string. This means that only characters will be output in the result string that match some character in the given set. For instance:

```

String s1 = "Subject age is 45 years";
String s2 = Strfilter(s1,"0123456789"); // Now s2 is "45"

```

Obviously, this function accepts also two Strings as arguments.

2.4.19 Strings in other languages

Strings are a very common, I would say almost universal data type. Here is a comparison with the interface provided by other languages.

<i>Language</i>	<i>Implementation</i>
C++	<p>The C++ strings package is very similar to the one proposed here. I used a similar interface on purpose: Avoid gratuitous incompatibilities with C++. Of course, essential differences remain. in C there are no constructors, destructors, templates etc.</p> <p>C++ accepts many initializers forms: you can write</p> <pre> string mystring("Initial content"); string mystring = "Initial_content"; string s2 = s1(mystring,0,1); // s2 is then "I" </pre> <p>Other operations are supported like append, insert, erase, replace, etc. They all use the C++ function table in each object:</p> <pre> s2 = "ABC"; s2.append("abc"); // Now s2 is "ABCabc" </pre> <p>The same operation in C is:</p> <pre> append(s2,"abc"); </pre>
Common Lisp	<p>Lisp has a "string" data type, that is defined as a specialized vector of characters. Accessing an element is done with:</p> <pre> (char "Abcd" 1) ==> "b" </pre> <p>Comparison is done with</p> <pre> (string= string1 string2) </pre> <p>There are other optional arguments to indicate a starting and ending point for the comparison. The result is either true (strings equal, or false, different). For lexicographical comparisons there are the functions <code>string<</code>, etc.</p> <p>To modify a character in a string you use:</p> <pre> (setf (char "Abcd" 1) #\1) ==> "1bcd" </pre>

<i>Language</i>	<i>Implementation</i>
APL	<p>Apl is a language where vectors are the main data type, in contrast to Lisp, where lists are paramount. APL strings are just vectors like all others. The rich primitive operations of APL in vectors can be applied to strings. For instance if A and B are strings of equal length</p> <pre data-bbox="549 422 632 447">A = B</pre> <p>will yield a boolean vector of equal length to A, filled with one or zeroes if the corresponding character positions match.</p>
Ruby	<p>Ruby strings can contain characters or just binary data, as our strings. There is an elaborate escaping mechanism to specify numerical values within strings. Embedded commands within strings allow specifications like:</p> <pre data-bbox="549 674 1046 732">"{ 'Ho! '*3}Merry Christmas" ==> "Ho! Ho! Ho! Merry Christmas"</pre> <p>Search expressions are very sophisticated, including regular expressions. There are more than 75 methods for the string library. Here is for instance the “slice” method:</p> <pre data-bbox="549 848 951 999">a = "hello there" a.slice(1) ==> 101 a.slice(1,3) ==> "ell" a.slice(1..3) ==> "ell" a.slice(-4..-2) ==> "her"</pre> <p>Note that slicing a string with only one index produces an integer, in this case the ASCII value of ‘h’.</p>
C#	<p>C sharp has a class string, with a similar implementation as the strings described above.</p> <p>C# strings are read only. If you want to modify them you have to use similar classes like “Buffer”. Normally you just make a new string when you modify some part of it.</p>

<i>Language</i>	<i>Implementation</i>
ADA	<p>Ada supports three kinds of strings:</p> <ol style="list-style-type: none"> 1) Fixed length. This length mustn't be known at compile time, it can be the result of a run-time calculation. 2) Bounded length. It can be used when the maximum length of a string is known and/or restricted. This is often the case in database applications where only a limited amount of characters can be stored. Like with Fixed-Length Strings the maximum length does not need to be known at compile time. 3) Unbounded length. <p>Here is an example of this type of strings in ADA:</p> <pre> with Ada.Text_IO; with Ada.Command_Line; with Ada.Strings.Unbounded; procedure Show_Commandline is package T_IO renames Ada.Text_IO; package CL renames Ada.Command_Line; package SU renames Ada.Strings.Unbounded; X : SU.Unbounded_String := SU.To_Unbounded_String (CL.Argument (1)); begin T_IO.Put ("Argument 1 = "); T_IO.Put_Line (SU.To_String (X)); X := SU.To_Unbounded_String (CL.Argument (2)); T_IO.Put ("Argument 2 = "); T_IO.Put_Line (SU.To_String (X)); end Show_Commandline; </pre>

2.5 String collections

After we had finished the strings library, we developed the string collection package. A string collection is a table of strings that will grow automatically when you add elements to it. It has a completely different interface as the strings library, using the popular

```
string.function
```

notation like in the C# language or in C++.

It uses an interface, i.e. a table of functions to provide the functionality and data access a string collection needs. Since the names of the functions are enclosed within the interface structure we can use mnemonic names like “Add”, etc, without fear of messing with the user name space, and without adding lengthy prefixes.

Other advantage of an interface are the extensibility of it. You can add functions of your own to the interface without interfering with the existing ones. We will discuss this later when we discuss subclassing, but it is obvious that you can define a new interface that has the first members as the given interface, but it has some extra members of your own.

2.5.1 The interface

We define first an empty structure, that will be fully defined later, to be able to define the functions in our interface

```
typedef struct _StringCollection StringCollection;
```

With his behind us, we can now define the interface:¹

```
typedef struct {
    // Returns the number of elements stored
    int (*GetCount) (StringCollection &SC);

    // Is this collection read only?
    int (*IsReadOnly) (StringCollection &SC);

    // Sets or unsets this collection's read-only flag
    int (*SetReadOnly) (StringCollection &SC,int flag);

    // Adds one element at the end. Given string is copied
    int (*Add) (StringCollection &SC,char *newval);

    // Adds a NULL terminated table of strings
    int (*AddRange) (StringCollection &SC,char **newvalues);

    // Clears all data and frees the memory
    int (*Clear) (StringCollection &SC);
};
```

-
1. Note that the interface uses extensively references, as here indicated by `StringCollection &SC`. This means that the argument must be a string collection object or a reference to it. This extension of `lcc-win32` allows us to avoid many test for a NULL pointer, and at the same time retain the efficiency of passing structure arguments using pointers insted of copying the value. To make this code work with a compiler that doesn't support references just eliminate the “&” and pass the structure by value. This will be discussed in more detail in the “Portability” section below.

```

//Case sensitive search of a character string in the data
bool (*Contains)(StringCollection &SC,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,char *SearchedString);

// Inserts a string at the position zero.
int (*Insert)(StringCollection &SC,char *);

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

//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 used supplied argument
// that can be NULL that 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,char *str);

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

// Replaces the character string at the given position with
// a new one
char *(*ReplaceAt)(StringCollection &SC,int idx,char *val);

```

```

// Returns whether the collection makes case sensitive
// comparisons or not
int (*IsCaseSensitive)(StringCollection &SC);

// Sets case sensitivity by comparisons
int (*SetCaseSensitive)(StringCollection &SC,int newval);

// Compares two string collections
bool (*Equal)(StringCollection &SC1,StringCollection &SC2);
} StringCollectionInterface;

```

Note that this lengthy structure is not replicated at each string collection object. Each string collection holds just a pointer to it, spending only `sizeof(void *)` bytes.

Once the interface is defined, the rest is quite simple:

```

// Definition of the String Collection type
struct _StringCollection {
    StringCollectionInterface *lpVtbl; // The table of functions
    size_t count;                       /* in element size units */
    char **contents;                    /* The contents of the collection */
    size_t capacity;                   /* in element_size units */
    unsigned int flags;                // Read-only or other flags
};

```

Each string collection will have a pointer to the interface. This uses a few bytes for the pointer, but I think this is RAM well spent. Besides, any medium size collection will hold a lot of data anyway, so the space used by the pointer is not significant at all.

The only exported function of this library is of course the creation function:

```
StringCollection * newStringCollection(int startsize=0);
```

The creation function allocates and initializes the data structure, setting a pointer to the interface function table.

2.5.2 Memory management

The easiest way to use the string collection is to use it with the garbage collector. The collection copies all the data it receives, but when indexing a collection you receive not a copy but the data itself. You have to remember to call the “Finalize” function to free the memory allocated by the collection when you are finished with it. And you should not free a string from the collection since this will provoke a trap. To avoid all this problems, just use it with the GC.

It could be argued that a simpler interface would have been to make a copy of the string that the user receives each time we access the collection but then, it would be necessary for you to free the strings received from the collection, what makes the usage of the library in a DLL that is called from another compiler runtime impossible.

2.5.3 Using the library

There is a problem however. To call one of those functions, for instance “Add”, to add a string to the collection we will have to write:

```

    SC = newStringCollection(25);
    SC->lpVtbl->Add(SC,"This is a string to be added");

```

Lcc-win32 has developed a shorthand notation for this, that allows you to write:

```

    SC->Add("This is a string to be added");

```

This is even more pronounced when indexing the string collection:

```

    char *p = SC->lpVtbl->IndexAt(SC,5);

```

instead of

```

    char *p = SC[5];

```

The algorithm used by the compiler is very simple. When dereferencing a structure, if the field indicated (in this case “Add”) does NOT exist, the compiler looks at the first position for a function table called “lpVtbl”. If this field exists, and it has a function called with the same name the user wrote (in this case, we know, it is the name “Add”), and this function pointer has as its first argument a pointer to the structure we are dereferencing, the compiler supplies the call as if the user had written the full syntax.

Here is a small sample program that demonstrates what you can do with this library:

```

#include <containers.h>
static void PrintStringCollection(StringCollection SC)
{
    printf("Count %d, Capacity %d\n",SC.count,SC.capacity);
    for (int i=0; i<SC.GetCount();i++) {
        printf("%s\n",SC[i]);
    }
    printf("\n");
}
int main(void)
{
    StringCollection SC = newStringCollection();
    char *p;
    SC.Add("Martin");
    SC.Insert("Jakob");
    printf("Count should be 2, is %d\n",SC->GetCount());
    PrintStringCollection(SC);
    SC.InsertAt(1,"Position 1");
    SC.InsertAt(2,"Position 2");
    PrintStringCollection(SC);
    SC.Remove("Jakob");
    PrintStringCollection(SC);
    SC.Push("pushed");
    PrintStringCollection(SC);
    SC.Pop();
    PrintStringCollection(SC);
    p = SC[1];
    printf("Item position 1:%s\n",p);
    PrintStringCollection(SC);
}

```

2.6 Implementation of the string collection

2.6.1 Creating a string collection.

```
StringCollection newStringCollection(int startsize=0)
{
    StringCollection result;

    memset(&result,0,sizeof(StringCollection));
    result.lpVtbl = &lpVtblSC;
    if (startsize > 0) {
        result.contents = MALLOC(startsize*sizeof(char *));
        if (result.contents == NULL) {

ContainerRaiseError(CONTAINER_ERROR_NOMEMORY);
        }
        else {
            memset(result.contents,0,
                sizeof(char *)*startsize);
            result.capacity = startsize;
        }
    }
    return result;
}
```

We allocate memory for the string collection structure, then for the data, and we set the capacity to the initial size. Since lcc-win32 supports functions with default arguments, we can suppose that a length of zero means the parameter wasn't there, and we use the default value. In another environments/compilers, the same convention can be used, but the argument must be there.

We are immediately confronted with the first design decision. What happens if we do not get the memory needed?

Many actions are possible:

- 1: Throw an exception. This can be caught with a `__try/__except` construct, and is a natural solution for exceptional conditions, in this case, there is no more available RAM.
- 2: Print an error message and abort the program. This is decision with too many consequences for the user, and gives him/her no chance to correct anything.
- 3: Return NULL. If there isn't any test in user code for a wrong return value, the program will crash the first time the user wants to use the collection. This is better than crashing in a library function, and such an error can be easily spotted in most cases. The problem with this is the complicated user interface. To catch all errors, a program must test for NULL reports at many places, what many people will not do.
- 4: Make all functions return a code indicating success or failure. The result would be stored in a pointer passed to the function, and no direct result would be available. This solution is implemented for example in Microsoft's strsafe functions.

In the first versions of the library, I had adopted solution 3: return NULL, no exceptions. But as the library code was developed, the testing for NULL became more and more cumbersome, and the code of the library grew with each NULL test. I decided then to adopt solution one, and throw an exception whenever a basic premise of the program wasn't sat-

ified: a patently wrong argument, and no more memory. I avoided throwing exceptions for errors that weren't critical, for instance trying to modify a read only container.

Note the usage of the FREE macro. We leave the user the choice of compiling the library with the GC or without it. If the macro NO_GC is defined, we avoid the garbage collector and use the malloc/free system.

We assign the table of functions field (lpVtbl) the address of a static function table defined in the module. Note that the function table can be in principle be modified by the user, by replacing one or more functions with other functions more adapted to his/her needs. This "subclassing" could be added later to the interface of the string collection library.

We have then in the header file:

```
typedef struct {
    int (*GetCount)(StringCollection &SC);
    //... and the prototypes of all other
    // interface functions go here
} StringCollectionInterface;
```

In our implementation file we define a master table like this:

```
static StringCollectionInterface lpVtblSC = {
    GetCount, IsReadOnly, SetReadOnly, Add,
    AddRange, Clear, Contains, CopyTo, IndexOf,
    Insert, InsertAt, IndexAt, Remove, RemoveAt,
    Finalize, GetCapacity, SetCapacity, Apply,
    Map, Push, Pop, ReplaceAt, IsCaseSensitive, SetCaseSensitive,
};
```

At each call of our constructor procedure we assign to the lpVtbl item this master table. Note that we do not copy the contents of the table, each StringCollection has a pointer to the original table. This allows for interesting side effects, as we will see later.

2.6.2 Adding items to the collection

This function returns the number of items in the collection after the addition, or a value less or equal to zero if there was an error.

```
static int Add(StringCollection &SC, char *newval)
{
    if (SC.flags & SC_READONLY)
        return -1;
    if (SC.count >= SC.capacity) {
        if (!Resize(SC))
            return 0;
    }

    if (newval) {
        SC.contents[SC.count] = DuplicateString(newval);
        if (SC.contents[SC.count] == NULL) {
            return 0;
        }
    }
    else
```

```

        SC.contents[SC.count] = NULL;
    SC.count++;
    return SC.count;
}

```

The library supports a “Read only” flag, that allows the user to build a collection and then protect it from accidental change. In the case that the flag is set we do not touch the collection and return a negative value to indicate failure.

If the collection is full, we try to resize it. If that fails, the result is zero and we exit. More about this below, but before continuing let’s see this a bit nearer

If the resizing succeeded or there was enough place in the first place, we go on and store the new element. Since the user could store a NULL value, we test for it, and avoid trying to duplicate a null pointer. Note that we always duplicate the given string before storing it

The function that resizes the collection is straightforward:

```

static int Resize(struct _StringCollection &SC)
{
    int newcapacity = SC.capacity + CHUNKSIZE;
    char **oldcontents = SC.contents;
    SC.contents = MALLOC(newcapacity*sizeof(char *));
    if (SC.contents == NULL) {
        SC.contents = oldcontents;
        ContainerRaiseError(CONTAINER_ERROR_NOMEMORY);
        return 0;
    }
    memset(SC.contents,0,sizeof(char *)*newcapacity);
    memcpy(SC.contents,oldcontents,SC.count*sizeof(char *));
    SC.capacity = newcapacity;
    FREE(oldcontents);
    return 1;
}

```

The algorithm here is just an example of what could be done. Obviously, if you resize very often the collection, the overhead of this is considerable. The new fields are zeroed¹, and the old contents are discarded.

2.6.3 Adding several strings at once

You can pass an array of string pointers to the collection, finished by a NULL pointer. The collection adds the strings at the end.

```

static int AddRange(struct _StringCollection & SC,char **data)
{
    int i = 0;
    if (SC.flags & SC_READONLY)
        return 0;
    while (data[i] != NULL) {

```

1. You see how this operation could be done faster? Look at the code again. Do we zero only the new fields?

```

        int r = Add(SC,data[i]);
        if (r <= 0)
            return r;
        i++;
    }
    return SC.count;
}

```

We use the Add function for this, iterating through the input string array. Note that any error stops the operation with some strings added, and others not. This could be done differently, making a “roll back” before returning, so that the user could be sure that either all strings were added, or none. To keep the code simple however, it is better to leave this “as is”.

2.6.4 Removing strings from the collection

RemoveAt returns the number of items in the collection or a negative value if an error occurs.

```

static int RemoveAt(struct _StringCollection &SC,int idx)
{
    if (idx >= SC.count || idx < 0 || (SC->flags & SC_READONLY))
        return -1;
    if (SC.count == 0)
        return -2;
    FREE(SC.contents[idx]);
    if (idx < (SC.count-1)) {
        memmove(SC.contents+idx,SC.contents+idx+1,
                (SC.count-idx)*sizeof(char *));
    }
    SC.contents[SC.count-1]=NULL;
    SC.count--;
    return SC.count;
}

```

We start by taking care of the different error conditions. Note that to simplify things several errors return the same result. A more detailed error reporting is quite trivial however.

If we are not removing the last item in the collection, we should move all items beyond the one we are removing down towards the origin. This is done with memmove.

We do not free the memory used by the pointer array, only the memory used by the string that we just remove. This could be a problem for a collection that grows to a huge size, then it is emptied one by one. We could add logic here that would check if the capacity of the collection relative to the used space is too big, and resize the collection.¹

A question of style is important here. Note that the code above *could* have been written like this:

1. Is it really necessary to set the pointer to NULL? Strictly speaking not, since the collection should never access an element beyond the count of elements. In a garbage collection environment however, setting the pointer to NULL erases any references to the string and allows the garbage collector to collect that memory at the next GC.

```
    SC.contents[--SC.count]=NULL;
    return SC.count;
```

instead of

```
    SC.contents[SC.count-1]=NULL;
    SC.count--;
    return SC.count;
```

We avoid a subtraction of one with the first form. But I think the second form is much clearer...

2.6.5 Retrieving elements

To get a string at a given position we use `IndexAt`.

```
static char *IndexAt(struct _StringCollection &SC,int idx)
{
    if (idx >=SC.count || idx < 0)
        return NULL;
    return SC.contents[idx];
}
```

What to do when we receive a bad index? This is a similar problem to memory exhaustion. Something needs to be done, maybe even more so than in the case of lack of memory. This is a hard error in the program. You can modify the code here to change the library, but keeping in line with the behavior of the library in other places, we just return `NULL`.

This is not a very good solution since the user could have stored `NULL` in the collection, at would be mislead into thinking that all went well when in fact this is not the case.

Since `lcc-win32` allows operator overloading, we can use it to easy the access to the members of the collection:

```
char *operator[] (StringCollection SC,int idx)
{
    return IndexAt(SC,idx);
}
```

We can now use the string collection as what it is: a table of strings

2.6.6 Finding a string

The function `IndexOf` will return the index of a string in the collection or a negative value if the string is not found.

```
static int IndexOf(struct _StringCollection &SC,char *str)
{
    int i;
    int (*cmpfn) (char *,char *);

    if (SC.flags & SC_IGNORECASE) {
        cmpfn = stricmp;
    }
    else cmpfn = strcmp;

    for (i=0; i<SC.count;i++) {
```

```

        if (!cmpfn(SC.contents[i],str)) {
            return i;
        }
    }
    return -1;
}

```

To avoid testing at each step in the loop if the comparison will be case insensitive or not, we assign the function to use to a function pointer before the loop starts.

2.6.7 Conclusion

These are the most important functions in the library. There are several others, and you can look at the code distributed with the lccwin32 distribution for the complete source. The documentation will tell you exactly what each function is supposed to do, and the different return values of each function. The source code will be installed in

`\lcc\src\stringlib`

It comes with a makefile, so if you have lcc in the path, you just need to type “make” to build the library.

2.7 Generalizing the string collection

We have now a general framework for handling string collections. Looking at the code, it is easy to see that with a little effort, we could make this much more general if we would replace the strings with a fixed size object, that can be any data structure. This general container is present in other languages like C#, where it is called “ArrayList”. You can store in an ArrayList anything, in C# it is not even required that the objects stored inside should be of the same type.

Since the nature of the objects stored is not known to the container, it is necessary to cast the result of an ArrayList into the final type that the user knows it is in there. In C# this is the “object”, the root of the object hierarchy, in C it is the void pointer, a pointer that can point to any kind of object.

If we look at the code of one of the string collection functions we can see the following:

```

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

```

We can easily generalize this to a container by changing the char pointer declaration to just void pointer!

Slightly more difficult is the Add function. The code of it in our string collection looked like this:

```

static int Add(StringCollection &SC,char *newval)
{
    if (SC.flags & SC_READONLY)

```

```

        return -1;
    if (SC.count >= SC.capacity) {
        if (!Resize(SC))
            return 0;
    }

    if (newval) {
        SC.contents[SC.count] = DuplicateString(newval);
        if (SC.contents[SC.count] == NULL) {
            return 0;
        }
    }
    else
        SC.contents[SC.count] = NULL;
    SC.count++;
    return SC.count;
}

```

We see that there are only two places where we have character string specific code: In the type declaration and in the call to the `DuplicateString` procedure. The first is easy to fix, but the second needs a little bit more reflection. `DuplicateString` doesn't need to know how many bytes to copy because it uses the terminating zero of the string to stop. When generalizing to a general data structure we can't count with the absence of embedded zeroes in the data, so we have to give the size of the object explicitly.

If we decide to store only one type of object into our containers, we could store the size in the container structure at creation time, and then use it within our code.

The code of our `Add` function for flexible arrays would look like this:

```

static int Add(StringCollection &AL, void *newval)
{
    if (AL.flags & SC_READONLY)
        return -1;
    if (AL.count >= AL.capacity) {
        if (!Resize(AL))
            return 0;
    }

    if (newval) {
        AL.contents[AL.count] =
            DuplicateElement(newval, AL.ElementSize);
        if (AL.contents[AL.count] == NULL) {
            return 0;
        }
    }
    else
        AL.contents[AL.count] = NULL;
    AL.count++;
    return AL.count;
}

```

This is almost the same code but now, we use the extra field in the container structure to indicate to `DuplicateElement` how big is the object being stored. Obviously,

DuplicateElement will use memcpy and not strcpy to copy the contents, but besides this minor change the code is the same.

The interface of the creation function must be changed also. We need an extra parameter to indicate it how big will be the objects stored in the flexible array. The interface for the creation function looks now like this:

```
ArrayList * newArrayList(size_t elementsize,int startsize=0);
```

The first argument is required and not optional: we must know the size of each element. The ArrayList structure will have a new field (ElementSize) to store the size that each object will have.

The change to other functions is trivial. Here is the code for the Contains function:

```
static int Contains(ArrayList &AL,void *str)
{
    int c;
    if (str == NULL)
        return 0;
    for (int i = 0; i<AL.count;i++) {
        if (!memcmp(AL.contents[i],str,AL.ElementSize))
            return 1;
    }
    return 0;
}
```

The code for the Contains function for a string collection is:

```
static int Contains(struct _StringCollection & SC,char *str)
{
    int c;
    int (*cmpfn)(char *,char *);
    if (str == NULL)
        return 0;
    c = *str;
    if ((SC.flags & SC_IGNORECASE) == 0) {
        cmpfn = strcmp;
    }
    else {
        cmpfn = stricmp;
    }
    for (int i = 0; i<SC.count;i++) {
        if (!cmpfn(SC.contents[i],str))
            return 1;
    }
    return 0;
}
```

Since we use memcmp, there is no sense to worry about case sensitivity. Besides that, only one line needs to be changed.

Note that we make a *shallow* comparison: if our structure contains pointers we do not compare their contents, just the pointer value will be compared. If our structure contains pointers to strings for example they will compare differently even if both pointers contain the same character sequence.

What is interesting too is that we use the same names for the same functionality. Since the function names are in their own name space, we can keep mnemonic names like Add, Contains, etc.

2.7.1 Subclassing

Let's suppose you have a structure like this:

```
typedef struct {
    int HashCode;
    char *String;
} StringWithHash;
```

You would like to change the code of the IndexOf function so that it uses the hash code instead of blindly making a memcmp.

This is possible by rewriting the function table with your own pointer, say, HashedIndexOf.

```
static int HashedIndexOf(ArrayList &AL, void *str)
{
    int i, h, top;
    StringWithHash *src = (StringWithHash *)str, *elem;

    h = src->HashCode;
    top = AL->GetCount();
    for (i=0; i<top; i++) {
        elem = (StringWithHash *)AL[i];
        if (elem->HashCode == h &&
            !strcmp(elem->String, src->String))
            return i;
    }
    return -1;
}
```

Before doing the strcmp, this function checks the hash code of the string. It will only call strcmp if the hash code is equal. This is an optimization that can save a lot of time if the list of strings is long. Note that *we kept the interface of the function identical* to IndexOf. That is very important if we want this to work at all.

You can set this function instead of the provided IndexOf function just with:

```
StringWithHash hStr;
// The structure is allocated, hash code calculated, etc.
// We allocate a 8000 long ArrayList and fill it in
ArrayList *AL = newArrayList(sizeof(StringWithHash, 8000);
// Code for the fill-in omitted
// Now we set our function at the right place
AL->lpVtbl->IndexOf = HashedIndexOf;
```

Done.

There is an important point to remember however. If you look at the creation code of the newArrayList function, you will see that the lpVtbl structure member is just a pointer to a static table of functions. By doing this you have made *all* ArrayLists use the new function,

even those that do not hold `StringWithHash` structures. You have modified the master copy of the function table. This may be what you want, probably it is not.

A safer way of doing this is to copy first the `lpVtbl` structure into a freshly allocated chunk of memory and then assigning the function pointer.

```
// Allocate a new function table.
ArrayListFunctions *newFns =
    GC_malloc(sizeof(ArrayListFunctions));
// Copy the master function table to the new one
memcpy(newFns,AL->lpVtbl,sizeof(ArrayListFunctions));
// Assign the new function in the copied table
newfns->IndexOf = HashedIndexOf;
//Assign this new table to this array list only
AL->lpVtbl = newFns;
```

This way, you modify only this particular `ArrayList` to use the function you want. Note that there is no freeing of the allocated memory since we used the memory manager. If you use `malloc` you should free the memory when done with it.

This subclassing is done automatically in other languages, but following specified rules. Here we are in C, you are on your own. This allows you much more flexibility since it is you that writes the rules how the subclassing is going to happen, but introduces more opportunities for making errors.

This method of replacing the default functions with your own will work of course with all structures having a function table, like the `StringCollection`.

2.7.2 Drawbacks

All this is nice but we have no longer compiler support. If we initialize an `ArrayList` with the wrong size or give an `ArrayList` the wrong object there is no way to know this until our program crashes, or even worst, until it gives incorrect results. We have used everywhere a void pointer, i.e. a pointer that can't be checked at all by the compiler.

2.8 Lists

Lists are a very popular data structure. You make one before you go to the supermarket, a list of things you need. A list is a series of items that can have any length and that is normally represented by a series of items in memory, joined by pointers from the first to the last (single linked lists).

We can describe this structure in C like this:

```
struct listElement {
    void * PointerToData;
    struct listElement *PointerToNextItem;
};
```

We have a first pointer that indicates just where the data item resides in memory, and another one, telling us where the next item can be found. It is customary to end the list by setting this pointer to a NULL value.

Going from the first item (the root of the data structure) to the last implies assigning a moving pointer to the first element, then following the “Next” pointers till we hit NULL.

```
struct listElement *p = Root;
while (p != NULL) {
    p = p->PointerToNextItem;
}
```

In single linked lists there is only one possible direction: forward. We can only start at the root and go to the last. To find the item “just before this one” we have to start again at the root and search forward until we find an item whose “Next” pointer points to this item.

This drawbacks can be solved with double linked lists, where we store two pointers, forward, and backward, each pointing to the next item and the previous item. This allows us to move from one item to the previous or next one, but comes at a price: for each data item we have now an overhead of two pointers and the pointer to the data, i.e. three pointers for each data item.

If the size of the item to be stored is smaller than the size of a pointer, we could optimize and save us an unneeded pointer by storing the data directly in the pointer slot.

Single linked lists are the simplest to begin with. We will design an interface for them, in the style of the interfaces of the string collection or the “Array list” containers. The interface will be very similar since the lists are also linear containers, there is a natural concept of order, or sequence, in them. It makes sense then, to use the array index operator to access this items, and to use the same names as we developed for the string collection and the ArrayList containers: Add, Insert, Remove, etc.

Of course there are fields that make no sense with lists. There is no need for a “Capacity” field, since we can store in a list as many items as we like, provided we do not ask for more items than our virtual memory system can hold. In modern machines this means an almost unlimited supply of items...

The same operations like “Pop” and “Push” will be implemented differently than in arrays. In the ArrayList structure, a Push operations adds an item at the higher indexes of the array, since the cost of getting to the last item is the same as the cost of getting into any other item. With lists, getting to the last element can be an expensive operation, so we just

add items at the first position and pop items from the first position. Adding items to the first position is very easy in lists, since we do not have to copy all items one place up as we would have to do with arrays.

The same as in the `ArrayList` structure we have now to decide if we want to store items of any type or just items of a single type. We decide for the later for “compatibility reasons”¹ We build for each list a small (as small as possible) header, containing information about this list.

```
struct _List {
    ListFunctions *lpVtbl;
    size_t ElementSize;
    size_t count;
    link *Last;
    link *First;
    unsigned Flags;
};
```

We store the root of the list in the “First” pointer, and to save us in many occasions a long trip through the list we store a pointer to the last item of the list. As usual there is a count field telling how many items we have, an `ElementSize` field telling us how big each item is.

Note that `First` and `Last` are pointers to a “link” structure. Here it is:

```
typedef struct _link {
    struct _link *Next;
    char Data[];
} link;
```

Instead of storing a pointer to the data, we store the data right behind the pointer. This is possible only when we know how big each item will be. When we allocate a link structure we allocate the size of the element + size of a pointer. With this organization we have the minimal overhead for lists that is theoretically possible: just one pointer. The overhead for storing items in a list is proportional to the number of elements in the list. For very long lists, storing 1 million items needs an overhead of 4 million bytes using this schema, 8 million bytes using a single link to the data, and 12 million for a double linked list.

2.8.1 Creating a list

There isn’t a lot of work to create the starting structure. Since there is no sense in getting an initial size, our constructor has only one obligatory element: the size of the elements to be stored in the list.

```
List *newList(int elementsize)
{
    List *result;

    if (elementsize <= 0)
```

1. Isn’t it a good “catch all” explanation? Actually it can mean: “I remain consistent with my bugs”

```

        return NULL;
    result = GC_malloc(sizeof(List));
    if (result) {
        result->ElementSize = elementsize;
        result->lpVtbl = &listlpVtbl;
    }
    return result;
}

```

We assign only two fields and rely on GC_malloc returning memory initialized to zero. We set the element size and the table of functions that this object will support. If there is no more memory left, or the element size is nonsense, we return NULL.

2.8.2 Adding elements

We use the same interface as the Add function in the ArrayList container.

```

static int Add(List *l, void *elem)
{
    if (l == NULL || elem == NULL)
        return LIST_ERROR_BADARG;
    if (l->Flags & LIST_READONLY)
        return LIST_ERROR_READONLY;
    link *newl = new_link(l, elem);
    if (newl == NULL)
        return LIST_ERROR_NOMEMORY;
    if (l->count == 0) {
        l->First = l->Last = newl;
    }
    else {
        l->Last->Next = newl;
        l->Last = newl;
    }
    return ++l->count;
}

```

We do not accept NULL elements in the list since it is a container that should hold elements of equal size, and a list of only NULLs doesn't make a lot of sense anyway.

We allocate a new link element to hold the data and a pointer. If the list is empty, this will be the first element and also the last one. We set the pointer accordingly. If the list is not empty, we add the new element at the end of the list by setting the "next" pointer of the last element to the new one, and then setting the last element to this one. We increase the count and we are done.

2.8.3 Retrieving elements

```

static void * IndexAt(List &l, int position)
{
    link *rvp;

    if (position >= (signed)l.count || position < 0) {
        ContainerRaiseError(CONTAINER_ERROR_INDEX);
        return NULL;
    }
}

```

```

    }
    rvp = l.First;
    while (position) {
        rvp = rvp->Next;
        position--;
    }
    return rvp->Data;
}

```

Here we should return a pointer to the data so we can't return an integer error code.¹ In case the index is wrong, we throw an exception. If the user has subclassed the exception procedure or the exception procedure somehow returns, we return NULL.

We set our roving pointer `rvp` to the first element, then we go through the elements until the given position is reached.

If the list has zero elements, this error will be caught in the line before with the condition:

```
position >= count
```

If the count is zero, position will be greater or equal than zero, since we catch the cases where the position is negative. The roving pointer `rvp` must be a valid pointer, and we should be able to go through the list until the desired position. Any error here will provoke a trap, and a trap here can mean only that we have an invalid list.

In `lcc-win32` we can overload the operator `[]` (indexing operator) with the same signature as `IndexAt`, to allow the user to write in the more natural notation

```
L[2]
```

instead of

```
L->lpVtbl->IndexAt(L, 2);
```

The advantage is just a matter of easy of use.

The declaration of the overloaded indexing operator would be:

```
void *operator [ ](List *l, int position);
```

1. We could have done that if we had kept the convention of never returning directly a result but a result code of the operation, as Microsoft proposed in the implementation of their `strsafe` library. The problem with that approach is that the interface is too cumbersome.