Acción de control FLC

Control Grueso

Control Fino

Selección de control fino error ≤ Banda de error

Variable control

Región de control fino

Banda de error

Tiempo
Real C++ programmers use templates. But for some, they are wasteful because they create a new copy of every algorithm for each class. Others will not use them because all the source code must be provided in header files.

1. A STACK CLASS

```
<template T> class Stack {
    int top; // private members
    T vec[100]; // constructed by T()
public:
    Stack(); : top(0) {}
    ~Stack(); {}
    void push(const T& it) : vec[top++]
        = it; |
    T pop() : return vec[--top]; |
};
```

Figure No. 1. A stack class

Consider class `stack`, shown in Figure No. 1. To instantiate a template, one just uses it to declare variables, as it is the case for both `Sint` and `Sfloat`.

Regrettfully, the result of using the template in Figure No. 1, to declare variables `Sint` and `Sfloat` is equivalent to defining two classes,

```
class Stack_int {
    int top; // private members
```

Figure No. 2. Duplicate code for a class

one for each type of stack, as shown in Figure 2. When more classes are used, as it is the case for the C++ Standard Template Library (STL), the worse this duplication becomes. In many applications the resulting executable programs are just a few kilobytes bigger, which are negligible for current computer memory sizes, but in others the size increase of the executable might be unbearable. But there is another problem with templates.

When a template class is used to declare a variable, the compiler specializes the template code for the declaration: this process is called
Instantiation; should the compiler not have the source code available, it would not be able to roll out the instantiated source code. For example, when variable `Sint` is declared in Figure 1, the compiler generates the equivalent to class `Stack< int>` in Figure 2. To replicate the class for the chosen type, enclosed in the square parenthesis `"<int>"` in the template invocation, the compiler needs the source code for the class, which forces programmers who implement templates to make all their algorithms public. At first sight this seems good to the people (quite a few claim that "software should be free"), but a closer look reveals some caveats.

When the programmer needs access to the source code, by mistake an error could be introduced. Hence, it is better for many different classes to share the same object code, which can be put into a library, a dynamic linked library (DLL), a shared library, or simply distributed in object form, ensuring that the programmer will not affect it. More software houses will be eager to develop generic algorithms when their intellectual property rights can be protected.

2. THE PROBLEM WITH POINTERS

Why did the C++ Standard Committee choose defining templates in such a manner to force the programmer to give away source code? Are there any ways to avoid this? The simpler answer is short: yes, just use pointers.

![Figure 3: A polymorphic list](image)

Many C++ class libraries implement reference containers, as the list shown in Figure 3. These contain not the values, but pointers to the values. The problem with this approach is that more space is required to store the container, because of the extra pointer used for each stored value. They also impose a time penalty to access stored values indirectly through their pointer. This is why this type of polymorphism was not built into C++; the alternative is to use templates, even though their use requires giving away the source code.

Reference containers are completely polymorphic, as they can contain any type of object, and even mixes of objects. They are quite popular as most compilers come with versions of these. The same object code is shared amongst all lists, regardless of the stored value type, and the client programmer just needs to know the specification to use them. But many C++ programmers do not like them. Recall that one of the design goals for C++ was to allow for the most efficient implementation for an algorithm, and using extra pointers compromises this requirement. That is why the STL was developed using templates: to achieve the efficiency of a hand-coded algorithm.

3. A COMPROMISE

When facing a problem, one usually tries to find a way to lessen the requirements to reach an acceptable solution. Complete optimality is seldom required. I decided to implement a list class with the following requirements:

1. It should be as efficient as the STL list class
2. Object code should be shared amongst all list classes
3. It should hide the list implementation
4. It should be programmed in C (not C++)

The first requirement is unavoidable, as otherwise people would no use the list class, preferring most of the times to either use STL’s or writing their own, to gain efficiency. C and C++ programmers crave for efficiency. The second requirement comes from my dislike for all the code replication that using the STL conveys. It bothers me that, when using a container, a full new copy of the implementation is deployed for each different data type.
While programming my list class, and after trying to meet the first two requirements, I found that the third one came for free. The fourth requirement is one of convenience, as many C programmers still will not use C++. Many object that "the C++ compiler available is difficult to use, which hinders software development", or "C++ bloats code", or "I have a native C compiler, but only a C++ cross-compiler", and even "I don't know C++".

I don't share any of these reasons, but they are valid to their defenders. But what really moved me is that, if it can be done in C, it will be done with plenty of elegancia in C++.

I believe that, when faced with scarcity, we have to squeeze every ounce of ingenuity to find a solution, which usually leads to a better overall product. That is why I like doing things the hard way, to get a better insight in how to achieve results, and oftentimes to find a more efficient solution. Recall that Alexander Stepanov, the STL's main architect, had the opportunity to change C++ to accommodate for his special needs [Ste-95]. Maybe a less favorable environment would have lead to a slimmer C++. Besides, the C language is ubiquitous.

What did I have to give in? The implementation I found works well for linked containers, as is the case for a linked list or a tree, but my solution does not go as far as reaching the efficiency and malleability of the sort() algorithm defined in header file <algorithm>, that comes in the STL library. Also, C++ templates can be a bit faster than my implementation, as templates can be used to get rid of function invocations.

4. DRAWING A LIST

Look into any data structures book, and you will find that lists are drawn as shown in Figure 4. If you examine closely the implementation in the STL library for the list class, which can be found in header file <list>, you will see that it follows this structure.

```
        elem_1 —&gt; elem_2 —&gt; elem_3 —&gt; NIL
           |            |            |
           |            |            |
          elem     next        elem     next        elem     next
```

Figure 4: Drawing a list

There is no problem with Figure 4, but compare it with the diagram in Figure 5. In this latter one, the arrows do not point to the boxes, usually called "nodes" in textbooks, but instead they link together the "next" fields, which I call "link fields". Is that a big difference? Not really, at least from the point of view of the code needed to implement the list algorithms, because in either list implementation the code juggles with the "next" field. If pointer "p" points to a node, this link field is accessed using the following syntax:

```
p-&gt;next
```

```
        elem_1       elem_2       elem_3       NIL
           |            |            |            |
           |            |            |            |
          elem     next        elem     next        elem     next
```

Figure 5: Drawing a list and its link fields

Why is this important? Because there are many algorithms that already exists for classes conceived as in Figure 4, which can be readily translated into equivalent algorithms for a class drawn as in Figure 5. As a matter of fact, what I did was to use a list implementation that I had shelved since my school days, to come up with the header file clist.h, shown in Listing 1, and its implementation clist.c, shown in Listing 2.
Instead of using the regular "point to the first" implementation, I decided to use a circular list (the "c" in "clist" stands for "circular"), as in Figure 6, because it is very efficient to add elements both at the beginning or the end of a circular list. Programmers seldom use them because they are a little more difficult to program; my hope is that my implementation will be reusable, so that never again a C programmer should have to invest time reprogramming this.

When implementing a list only dealing with link fields, it does not matter at all what it is the element type contained in the list, because every algorithm only deals with the linked fields. However, some pointer shifting is required to transform a pointer to the link field into a pointer to the value stored in the list.

6. C IMPLEMENTATION

In the header file clist.h two important types get declared. Structure link_rep is the link field used to implement the list, and clist_rep is the list itself, which contains the pointer to the last element in the list. These names end in "rep" because they are the internal Representation of the data types. To fake data hiding a little, I used the interface structures called cl_link and clist. In the header files, all the operations for these abstract data types follow their declaration.

C does not support neither encapsulation nor name overloading, which forces the programmer to use naming conventions to fake those C++ features. What I chose to do is to prepend every operation name with the letters "cl_", which stand for circular list. For the link field type, cl_link, only two operations are required: initialization with cl_link_init() and destruction with cl_link_done(). These names end with an underscore "_" because I also provide macros to inline their implementation, named without the underscore (they appear towards the end of header file clist.h). As the C language does not support references, these routines receive pointers to link fields.

The operations for the circular list are the usual: initialization, destruction, copying, counting, etc. As it happens for link fields, those names that end in underscore can be inlined using the corresponding macro. What is significantly different is the way to store values in one of these lists: instead of "inserting" the value, what one should do it to "link" it to the list. Hence, there are no "cl_insert()" or "cl_delete()" operations, but "cl_link_after()" and "cl_unlink_after()". These take as arguments pointers to the link fields that will get stored within the list.

7. USING LISTS

```c
typedef struct {
  int n;
  cl_link ik;  /* link field */
  int lk;
} clist;
```

Figure 7: int lk

These lists force the client programmer to include the link field in any structures that will be stored in a list. For example, should a list of integers be needed, then the programmer must come up with a structure that includes the integer value and the link field for the list, as it is done in Figure 7 (the complete declaration...
and implementation for this type are shown in Listing 3 and Listing 4). How is it to program using these lists? The quicker answer is to examine the implementation of function primes(), in the file c-list.c shown in Listing 5 (notice the dash "-" in the name; c-list.c is the main test program). Before storing a value in the list, a new linkable structure must be obtained from dynamic memory, and later linked into the list. In function primes(), the pointer "pInt" is used to create a new value invoking the macro binder_create(), which is defined in header file "binder.h", shown in Listing 6.

The result of unrolling

```
int lk* pInt;
binder_create(Bi, pInt)
```

to avoid ambiguity when the macro gets unrolled.

Where did variable "Bi" get defined? In the main program file c-list.c. It is a constant pointer, that can not modify what it points to. Look for the following line:

```
const binder * const Bi -
&name_binder(int lk, ik);
```

The macro name_binder(int lk, ik) is also defined in header file binder.h. It is used to build the name of the structure that contains both the "Bi->size" field and the functions pointer "Bi->init()", which are declared in file intrat.h and defined in intrat.c (just in case you forgot the difference between "defining" and "declaring", remember that you put declarations in header files, and implementations -definitions- elsewhere; it is usual for C definition files to have names that end in ".c").

Look closely again to the macro invocation for name_binder(int lk, ik): it contains the name of the type, "int lk", and the name of the field used to link it to a list "ik". Hence, this macro comes up with the name "Binder_int lk ik", which gets declared and initialized in intrat.c. "Bi" is just a shorthand that references this structure.

The macro define_binder() is invoked in the implementation file intrat.c, and it gets passed the addresses of each function used to handle structure int lk; in particular, it gets a pointer to int_init(), the function that carries out the duty of initializing the list node. In other words, "Bi->init()" executes function int_init().

After creating the new node, its value "pInt->n" is updated, and later the whole node is linked into the list:

```
c_l_link_after(L, pHere, &pInt->lk);
```
use wrapper macros, or wrapper C++ templates.

As the compiler has no recollection of what type of elements are stored in a container, it is up to the programmer to keep track of this important fact. The function `has_digit()` traverses a list that contains numbers; to use them, the programmer must resort to macro `intlk_cast(p)` to transformer a position in the list, which always is a pointer to a link field (type `cl_link*`), into a pointer to the whole node (type `intlk*`). This is the purpose of defining `intlk_cast(p)` in header file `intranat.h`; the result of unrolling its invocation is to obtain the following code:

```c
binder_cast(p, intlk, ik)
```

This is, in turn, the invocation of another macro, `binder_cast()`, defined in header file `binder.h`, which gets unrolled to yield this:

```c
{(*intlk*)[void*]{(*char*)[p]}-
   offsetof(intlk, ik))}
```

This code begots some explanation. Lists are implemented having link fields pointing to one another. The client programmer does not care for those pointer, but for the values stored in the list. Hence, there must be a way to change a pointer to the link field into a pointer to the complete value. The procedure to achieve this goal is to subtract from the link field pointer the offset at which the link field appear in the node to get, for example, `pInt` from `pHere` in Figure 8. This offset can be obtained with the C standard macro `offsetof()` defined in header file `stddef.h`.

Why is pointer "p" in macro `SUB_OFFSET()` typecasted into a `(char *)`? Because the C standard clearly states that a "char" always has a size of one, which ensures that the pointer arithmetic used to adjust the link field pointer yields a pointer to the first bit in the stored value. Again, the extra parenthesis are there to avoid ambiguity; they are annoying only when looking at the source code after the
C preprocessor mangles it, which is seldom needed. Looking back into Figure 8 it is by now clear that all these pointer juggling just "adjusts back" the pointer to the link field into a pointer into the full stored value.

What is the difference between polymorphism and parametrization? According to the experts [MDC-91], the first is a more general concept, whereas the later is a special type of polymorphism usually called textual polymorphism. For most people, parametrization is a synonym for generic programming. C++ supports parametrization in the form of textual polymorphism; C++ is not a fully polymorphic language.

9. POLIMORPHIC FUNCTIONS

You can argue that containers implemented linking fields contained in their elements are not polymorphic, at least no in the traditional ways. However, what matters is whether is it possible to write functions that process all the values in a container, regardless of the stored value types. An example of such a function is the operation cl_print_binder(L,F,B), defined in clist.c. This function receives three parameters:

```c
const clist * L
```

The usual pointer to the container.

```c
FILE * F
```

The file where to print the contents of the list.

```c
const binder * B
```

A binder, where information about the node stored in the list is recorded.

It is in the binder where all the information to deal with a value stored in a list is collected. The B->size is needed to malloc() a new node of the right size. B->offset is needed to adjust a link pointer into a node pointer. The other fields defined in binder.h are pointers to functions that provide basic operations like copying, initialization, etc. One of these is the pointer to the print() operation, called int_print() for the int lk type.

The implementation of the polymorphic print() operation for the list is straightforward. The list traversal starts at the first node, obtained invoking operation cl_first(). Each time that a stored value needs to be printed, the print function is invoked through the print() function pointer stored in the binder: B->print(). Some heavy pointer typcasting is required, but a neat trick is to recall this pointer into a local variable, "PRINT", that can be used to clearly make the function invocation:

```c
const void (*)(* PRINT)(void *, FILE *)
```

```c
=B->print;
/* ... */
PRINT( SUB_OFFSET(p, ofs), F );
```

The main difference between this code and the one used in function primes() is that in here we really do not care for the type of the values stored in the list, because we just need to print them, and that task is carried by operation B->print(), whereas in the other case we had to typcast back to a node pointer each list position to use it in the algorithm. Hence, there are two different programming styles: generic programming, when an algorithm works on a list regardless of the stored value types, and client programming, when one uses the container to store, retrieve and use values.

The efficiency of operation cl_print_binder() matches that of a hand coded function, because no indirect pointer jumps are used. As a matter of fact, variable PRINT is used to avoid looking up this pointer in the binder B. As both cl_first() and cl_last() are inlined macros, the object code generated for this algorithm is as fast as the one generated should the STL library were being used in C++. The difference here is that all the implementations are sharing the same algorithm, and the price paid is just some pointer adjustment used to
call the stored value print() operation. Nonetheless, let us see why the STL would generate faster code than this.

```c
void qsort(
    void* base, size_t nelem, size_t width,
    int (*fcmp)(const void *, const void*)
);
```

**Figura 9. qsort()**

If you delve into the standard C header `<stdlib.h>` you will find the qsort() ordering function, shown in Figure 9, which receives a function pointer, fcmp(), that it uses to compare the elements in array base until it is left ordered. The same object code will work to sort an array of any type; most compiler vendors only provide the object code to qsort(), and the programmer just needs to know its specification to use it.

However, each time that qsort() compares two elements, it needs to incur in a function call when invoking fcmp(), which slows down execution. In contrast, the generated code for the STL sort algorithm will avoid such call by inlining the implementation of the fcmp(). From this it follows that, when the print() operation is not implemented as an inline function, the speed of the STL print() method for list would be the same as that of operation cl_print_binder().

### 10. LISTS OF LISTS

There is another test that a generic container must pass. It should be possible to store a whole container inside another. To try this for circular lists, I devised function PI_listlist() that receives a string and creates a list containing as many list as defined by the numbers in the string. For example, for string "3.14159" it should print the following list of list:

```plaintext
( [ 3 3 ] [ 1 ] [ 4 4 4 ] [ 1 ] [ 5 5 5 5 ] [ 9 9 9 9 9 9 9 ] )
```

The first thing to do, as it was the case for type int lk, is to create a new "list of lists" type, called Lint and defined in the test program itself, c-list.c. What can be stored in a Lint variable is a list, but every Lint can itself be put into a list. Hence, the list of lists "LL" is not of type Lint, but just a regular list that just happens to contain Lint values.

What is delicate is creating a node to store in LL. For example, for digit '3', a list with three integer values must be created, and linked into LL, to obtain its first value: (3 3 3). This value is created using pointer "L" and invoking macro binder_create(). However, contrary to what was done in function primes(), when creating another Lint value to store in LL, the real name for the Lint binder is used: "&name_binder(Lint, lk)". That is why this binder needs to be declared, at the very beginning of test program c-list.c:

```c
declare_binder(Lint, lk);
/* Binder_Lint_lk */
```

The new node that pointer "L" denotes is already initialized, because macro binder_create() invokes the initialization function for the Lint type: Lint_init(). After this all its values are generated in a for loop. The code to create and store a value in this inner list is similar to that used in function primes():

```c
int lk*pInt;
binder_create(Bi, pInt)
```

What changes is the way to append a value: cl_append(&L->L, &pInt->lk);

Why is there a difference? Recall that "L" points to a node that can be linked into the list of lists "LL", so that L->L is the list inside "L". The ampersand "&" is used because operation cl_append() expects a pointer to a list. How does the big list LL get its value printed? Its enough to invoke:
cl_print_binder(LL, stdout, &Binder_Lint_1k);

11. LINK FIELD INITIALIZATION

Values that will be stored in generic lists require to be defined as structures that contain a link field. This field, nonetheless, does not belong to the value, but to the list. Hence, the operations for the stored value should never use nor change a link field. Right? Wrong! Let us see why.

Look into the implementation of the initialization operations for types int_1k and Lint. Both of these operations, int_init() and Lint_init() initialize the link field, called "1k" and "lk" in each case, invoking the list link constructor "cl_link_init()". The destructors for these same types also destroy these link fields. Why is that?

The answer is simple: upon creation, a list node is just raw memory. After initialization, all its fields must contain consistent values. Upon creation of a list node, it makes sense to initialize every field within, including those that belong to the container. But never again should it be touched only when destruction should take place. Doing things another way will force to include complicated machinery within both the container and the binder implementation, which is messier.

Nonetheless, no other operations but the constructor and destructor should change or use the link field. For example, the copy operations for both int_1k and Lint never access the link field: both Lint_copy() and int_copy() skip over it.

That was why link fields should be initialized and destroyed by the node's constructor and destructor. However, a close examination of the implementation of both cl_link_init() and cl_link_done() reveals that their inline code is empty, or somehow weird when macro NDEBUG is not defined. The idea behind this is the following: for debugged code, and in the case of the list container, no initialization should be done for the "next" link field. However, when debugging, it is nice to have the compiler issue warnings if things are not done the right way.

For other containers, for example a tree or a graph, node initialization is not as trivial as for the list. For example, when using the list container to implement a graph, the graph link field would be a list, and the gr_link_init() constructor would have to invoke the list initializer cl_init(). A similar thing would be required for its destructor.

The standard C NDEBUG macro is also used in the implementation of the list operations cl_link_after() and cl_unlink_after(). In the first case, the conditional compilation includes code to check whether an already linked node is being used, because it contains a non NULL value, or whether an unlinked node is being decoupled from the container, when the link field is already NULL.

12. EVALUATION

How different is to program using the clist generic list instead of a STL list, or a hand coded one? Most of the time, the client programmer will code algorithms as found in the implementation of functions primes() and digits() in Listing 5.

The main differences here, should you not want to call then "annoyances", is to define the data type int_1k to include a field, and to invoke macros binder_create() or binder_cast() to change "position in the list" pointers into "value" pointers. The rest of the code is similar to that used in regular lists. What if one needs to code a generic algorithm?

Examine again the implementation of the cl_print_binder() function. To come
up with such code you need to understand a little bit more how binding is done between the container and its contained value: the main idea is that link field pointers need to be adjusted back to point to the beginning of the each node before using the operations stored it the binder table (see Figure 8). However, most programmers will not need to code generic algorithms.

Let us examine this implementation from the optics of the requirements stated before:

It should be as efficient as the STL list class

Almost there: as templates can avoid function invocations of the contained type, template instantiation for simple types (like char, int, long, etc.) can result in faster code. For most containers both this implementation and the STL would invoke functions to carry out the basic operations of initialization, copying, etc. When dealing with linked containers some speed savings will result because their values need not exist in dynamic memory always.

Object code should be shared amongst all list classes

Mission accomplished!

It should hide the list implementation

Mission accomplished!

It should be programmed in C (not C++)

Mission accomplished!

To my eyes, this is success: you will be the judge.

13. TRY DOING THIS!

It would be nice if there was something that could be done with “link” container but not with regular containers. There is one thing: suppose you need a value to be stored in two different containers, say a tree and a list. When using linked containers the way to do it is to include two link fields in the node, one for the tree and the other for the list. Can the same thing be done with regular, STL like, containers?

The answer is no, unless references (pointers) to the values are used. This follows from the fact that STL containers keep their stuff in dynamic memory, where they copy or remove values when the insert or delete operations are used. Hence, what gets stored in a container is always a copy of the original value, and when trying to store the same thing in two places the result will be to have two copies, leaving the original untouched: that is not the best for all occasions.

14. CONCLUSION

You can argue, quite convincingly, that using binder.h is a mess because a lot of new data types should be defined just to link them into a list. I can answer back that it is a good programming practice to encapsulate data types in their own structure, but you can counter attack and say that efficiency is not only measured in terms of program speed and time usage, but also in terms of programming easy of use. I could counter argue that when a list is just another service of the operating systems, or the software platform, then programmers do things right faster, after they learn how to use generic containers. We could keep arguing back and forth, but at last I would say this: if you read up to here, then maybe the ideas I threw at you are not that bad. Give it a try, and let me know what the experience of using this approach teaches you. Remember that most ideas are useless, and many are useful just to breach the gap to reach other discoveries.

A little macro tweaking with some pointer juggling yields linked containers good enough for most applications. It is always better to implement them in an Object Oriented Language [Str-88], but with a little care a programmer can build a container library in C [BSG-92] that is efficient and provides some of the better features found in more complicated libraries, like the C++ STL. You
15. ACKNOWLEDGMENTS

Both Ronald Argüello and Carlos Loria took the time to criticize and earlier version of this paper. This research was made in project 326-98-391 "Polimorfismo uniforme más eficiente", funded by Vicerrectoría de Investigación in the Universidad de Costa Rica. The Escuela de Ciencias de la Computación e Informática has also provided funding for this work.

16. BIBLIOGRAPHY


http://www.sgi.com/Technology/STL/drdobbs-interview.html

Este artículo está disponible en el siguiente sitio internet: http://www.dimare.com/adolfo/p/c-list.htm
ANEXO 1

```c
/* clist.h v0.1 (C) 1999 adolfo@dl-mare.com */

#ifndef CLIST_H
#define CLIST_H

#include "binder.h"
#endif

#ifdef __cplusplus
extern "C" {
#endif

typedef struct link_rep {
    struct link_rep * next;
} link_rep;

typedef struct {
    link_rep * last;
} clist_rep;

typedef struct {
    link_rep private; /* private */
} cl_link;

typedef struct {
    clist_rep private; /* private */
} clm,

typedef *clist;

void cl_link_init (cl_link* lk);
void cl_link_done(cl_link* lk);

void *cl_init_ (clist *);

void cl_link_after(clist *, cl_link*, cl_link*);
cl_link* cl_unlink_after(clist *, cl_link*);

c_link* cl_first_ (const clist *);
cl_link* cl_last_ (const clist *);
cl_link* cl_next_ (const clist *, cl_link *);

c_link* cl_nth(const clist *), cl_link *q, size_t n);

size_t cl_count (const clist *);
int cl_empty_ (const clist *);

void cl_append_(clist *, cl_link *);

/* All these require a binder */

void cl_swap_binder (clist *, clist *, src, const binder *);
void cl_copy_binder (clist *, clist *, src, const binder *);
int cl_equal_binder(const clist *, const clist *,
const binder *);
void cl_print_binder(const clist *, FILE *,
const binder *);
void cl_delete_all (clist *, const binder *);
void cl_done_binder (clist *, const binder *);

#ifdef NDEBUG
    /* Use#define to optimize out code */
    #define cl_link_init(lk)
    #define cl_link_done(lk)
#else
    /* typecheck, but only a little */
    #define cl_link_init(lk) ((link_rep*) lk)->next = NULL
    #define cl_link_done(lk) do {} while( (lk) != (lk) )
#endif

#define cl_init(L) ((clist_rep*)(L))->last = NULL
#define cl_first(L) (cl_link*)
    
    { NULL = ((clist_rep*) L)->last }

?
```
```c
/* c-list.c v0.1 (C) 1999 adolfo@di-mare.com */

#define NDEBUG /* production code should
#include "intran.h"

typedef struct {
  clist_lk; /* list of int lk */
  Lint;}

declare_bind(Lint, lk);
  clist lk - * const Bi - &name_bind(int lk, ik);
  Lint_copy(Lint *L, Lint *src) {
    cl-copy_bind(L<--L, *src<--B, B);
    } int Lint_equal(Lint *L, Lint *src) {
    return cl_equal_bind(L<--L, *src<--B, B);
    } void Lint_print(Lint *L, FILE *F) {
    cl-print_bind(L<--L, F, B);
    } void Lint_done(Lint *L) {
    cl-done_bind(L<--L, B);
    cl-link(L<--L, B);
    }

define_bind(Lint, lk,
Lint_init, Lint_copy, Lint_equal, Lint_print,
Lint_done);

#define FALSE 0
#define TRUE (!FALSE)

void primes(clist *L, unsigned n) {
  /* Ppends to L the prime numbers smaller than "n" */

  int i, j;
  Lint lk = * NULL; /* points to pInt->ik */
  for (i = 1; i < n; i++) {
    int is_prime = TRUE;
    for (j = 2; j < i; j++) {
      if (0 == i % j) {
        is_prime = FALSE;
      }
    }
    if (is_prime) {
      int lk * pInt; /* pointer to the node */
      binder_create(Bi, pInt);
      pInt->n = i;
      cl-link after(k, phere, 4pInt->ik);
      phere = 6pInt->ik; /* current position */
    }
  }

  void has_digit(clist *L, int d, FILE *F) {
  /* Print all numbers in L that have digit "d" */
```
INGENIERÍA

```c
#include "string.h" /* strlen() */
void PI_listlist(const char *V) {
  Uses LL, a list of lists, and prints it
  
  int i, j;
  size_t len = strlen(V);

  clist LL; /* list of lists */

  cl_init(4LL);
  
  fprintf(stderr, \n"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
```c
#include <stdlib.h>

void P1_listlist(Const char *V) {
    Uses LL a list of lists, and prints it
}

int i, j;
size_t len = strlen(V);

cList LL; /* list of lists */

cList LL;
for (i = 0; i < len; i++) {
    int W = ('.' - 'V[i]') ? 'V[i]'- '0';
    List *L;
    /* add a new element to LL */
    binder_create(1, name binder(int, 1k, L));
    cList append(LL, LL->lk);
    for (j = 0; j < W; j++) {
        int lk * pInt;
        binder_create(1, pInt);
        pInt->om = W;
        cList append(LL->lk, pInt->lk);
    }
    fprintf(stdout, ",");
    cList print binder(4L, LL, stdout, 1k);
    fprintf(stdout, "n");
    cList print binder(4L, LL, stdout, 1k);
    fprintf(stdout, "n");
    cList done binder(4L, name binder(int, 1k));
}

int main() {
    cList L;
    unsigned long memAvail = coreleft();
    cList init(4L);
    primes(4L, 102);
    fprintf(stdout, "minprimes(102) == \n");
}
```

**ANEXOS**

```c
#include <c-list.h>

void cList after(cList *L, cList *p, cList *px) {
    /*
    Links after position "p" the object that contains
    link field "px".
    */
    To link as first set p == NULL
}

void *cList after (cList *L, cList *p, cList *px) {
    if (NULL != px->next) { /* check unlinked */
        abort();
    }
    if (NULL == L->last) { /* empty list */
        L->last = px;
        px->next = NULL;
    } else if (NULL == p) { /* link as first */
        L->last = px;
        L->last->next = px;
        px->next = px;
    } else if (p == L->last) { /* link as last */
        L->last = px;
        L->last->next = px;
    }
}
```
L->last = px;
     
else
  
  /* link in the middle */
  p->next = p->next;
  p->next = px;
}

cl_link cl_unlink_after(clist *L, cl_link *p) {
  
  /***************************/
  /* Detaches from the list the object that comes after */
  /* the object pointed to by the argument p. */
  /* In case L->last = NULL, */
  /* it returns a pointer to the link field just detached. */
  /***************************/
  
  clist_rep *L = (clist_rep *) L;
  link_rep *p = (link_rep *) p;
  link_rep *p;

  ifndef NDEBUG
  if (NULL == p->next) { /* check unlinked */
    abort();
  }
  endif

  if (p == NULL) {
    px = L->last->next;
    if (L->last == L->last->next) { /* just 1 element */
      L->last = NULL; /* detach the first */
    }
    else
      p = L->last->next;

    L->last = p;
  }

  else /* p != NULL */
    px = p->next;

  if (L->last == L->last->next) {
    abort(); /* only 1 element */
  }

  else /* detach from the middle */
    
    link_rep *tmp;
    tmp = p->next;
    p->next = tmp->next;
    if (tmp == L->last) L->last = px;

  endif

  ifndef NDEBUG
  px = NULL; /* clean up link field */
  endif
  return (cl_link *) px;
}

cl_link cl nth(const clist *L, cl_link *q, size_t n) {
  
  /***************************/
  /* Returns the position of the "n"th element */
  /* counting from "q" */
  /* For n = 0 returns "q" */
  /***************************/

  size_t i;
  link_rep *p;

  if (0 == n) {
    return q;
  }
  else if (NULL == q) {
    p = (clist_rep *) L)->last;
  }
  else {
    p = (link_rep *) q;
  }

  for (i = 0; i < n - 1; i++) {
    p = p->next;
  }

  return (cl_link *) p;
}

size_t cl count(const clist *L) {
  
  /***************************/
  /* Returns the number of elements in the list */
  /***************************/
  
  clist_rep *L = (clist_rep *) L;
  link_rep *p = L->last;

  size_t n = 0;

  if (L->last == NULL) {
    p = L->last;
    while (p != L->last)
      n++;  
  return n;
}
}

void cl swap (clist *LL, clist *SRC, const binder *B) {
  
  /***************************/
  /* Swaps the contents of "LL" and "SRC" */
  /* - No copying in done: only pointer swapping */
  /* - Takes O(1) time and space */
  /***************************/

  link_rep *TMP = (clist_rep *) (LL)->last;
  (clist_rep *) (LL)->last = (clist_rep *) (SRC)->last;
  (clist_rep *) (SRC)->last = TMP;

  strncpy a; /* space acquired */

  void cl copy (clist *LL, clist *SRC, const binder *B) {
  
  /***************************/
  /* Copies list "src" over "L" */
  /* - B->copy() is used to copy each element */
  /* - List "src" remains unchanged */
  /* - Takes O(n) time and space */
  /***************************/

  clist_rep *L = (clist_rep *) L;
  clist_rep *src = (clist_rep *) SRC;
  link_rep *DELETED;

  if (L == SRC) {
    return; /* avoid auto-copy */
  }

  DELETED = L->last;
  if (NULL == DELETED) {
    L->last->next = NULL;
    L->last = NULL;
  }

  /***************************/
  /* copy element by element, from src */
  if (NULL == src->last) { /* src isn't empty */
    const offs = B->offset;
    const size = B->size;
    const void (*INIT) (void *); B->init;
    const void (*COPY) (void *, void *) = B->
    copy;
    /***************************/
    /* INIT: pointer to a function that returns */
    void;
  }

  else /* p SRC */
    
    link_rep * pSRC;

  pSRC = SRC->last;
  do {
    void * psrc;

  }
link_rep + plink; /* == plnow->link */
psrc = plsrc->next;
*/ /* 'bend backwards' to avoid malloc()-ing */

if (NULL != DELETED) /* reuse nodes from L */
  plink = DELETED;
  plnow = SUB_OFFSET(plink, ofs);
  DELETED = plink->next;
  while (psrc != src->last);

  if (NULL != DELETED) /* delete rest */
    const ofs = B->offset;
    const void (* DONE) (void *) = B->done;
  while (DELETED != NULL) {
    void *del = SUB_OFFSET(DELETED, ofs);
    link_rep *p = DELETED->next;
    DONE(del);
    freedel();
    DELETED = p;
  }

  }

int cl_equal_binder(const clist *L, const clist *src,
const binder *B) {
  //====================================================================
  // Returns 0 when L and src are different
  // - B->equal() is used to compare elements
  // - Takes O(n) time and O(1) space
  //====================================================================

  link_rep *p, *q;
  link_rep *L = (clist_rep *) L;
  link_rep *src = (clist_rep *) src;
  const ofs = B->offset;
  const int *EQUAL = B->equal();
  if (L->last == src->last) {
    return 0; /* TRUE */
  }

  if ((NULL == L->last) || (NULL == src->last)) {
    /* avoid using a NULL pointer -dereferencing- */
    return 0; /* FALSE, because both can't be NULL */
  }

  /* compare elements one by one */
  p = L->last;
  q = src->last;
  do { /* !EQUAL(SUB_OFFSET(p, ofs), SUB_OFFSET(q, of$s)) */
    return 0; /* FALSE */
  } while (p != L->last) && (q != src->last);

  return (p == L->last) && (q == src->last);
}

void cl_print_binder(const clist *L, FILE *F, const
BINDER * B) {
  //====================================================================
  // Prints the whole list with commas and |
  // parenthesis, like this: |
  // - (1, 2, 3, 4, 5) or even (1) |
  //====================================================================

  const ofs = B->offset;
  const void (* PRINT) (void *, FILE *) = B->print;
  cl_link *last = cl_last(L);
  
  while (NULL != p) {
    PRINT(SUB_OFFSET(p, ofs), F);
  }

  cl_link *p = cl_first(L);
  printf(F, "\n");

  while (NULL != p) {
    PRINT(SUB_OFFSET(p, ofs), F);
    if (p != last) {
      printf(F, " ,\n");
      p = cl_next(L, p);
      printf(F, "\n");
    }
  }

  cl_delete_all(L, B); /* Destroys the whole list */

  while (L->last != NULL) {
    void *del = (char *)L->last - B->offset;
    B->done(del);
    freedel();
    /* after free() */
    L->last = p;
  }

  cl_done_binder(L, B);

  // Non-macro versions of inlined code */
#define COMPAT_NON_INLINE Methods

  void cl_link_init(cl_link *link) {
    #pragma argued
    /* avoid "argument [link] not used" warning */
  }

  void cl_link_done(cl_link *link) {
    #pragma argued

}
void _c_init (clist *l) {  
  // Initializes the list  
  (clist_reg) (l)->last = NULL;  
}  

cl_link *cl_first (const clist *l) {  
  return cl_first (l);  
}  

c_link *cl_last (const clist *l) {  
  return cl_last (l);  
}  

c_link *cl_next (const clist *l, cl_link *p) {  
  // Returns the position that comes after "p"  
  // - After the last position, returns NULL.  
  return cl_next (l, p);  
}  

int cl_empty (const clist *l) {  
  return cl_empty (l);  
}  

void _c_append (clist *l, cl_link *p) {  
  // Appends to the list  
  cl_append (l, p);  
}  

#define _EOF  
/* EOF: clist.c */

ANEXO 6

/* intrat.c 0.1 [C] 1989 adolf@d1-mare.com */

#include "intrat.h"

typedef int lk,  
int_init, int_copy, int_equal, int_print,  
int_done  
;

void int_init (int lk * it) {  
  it->num = 0;  
  cl_link_init (it->lk);  
}  

void int_copy (int lk * it, int lk * src) {  
  it->num = src->num;  
  cl_link_init (it->lk);  
}  

int int_equal (int lk * it, int lk * src) {  
  return it->num == src->num;  
}  

void int_print (int lk * it, FILE *f) {  
  printf (f, "lk=%d", it->num);  
}  

void int_done (int lk * it) {  
  cl_link_done (it->lk);  
}  

/*******  
** rat lk  
*******/*

#define _EOF  
/* EOF: intrat.c */