

ABL

(Abstract Basic List) 1.0

---

Anton Kulchitsky ([anton@kulchitsky.org](mailto:anton@kulchitsky.org))

---

This manual is for ABL (Abstract Basic List) (version 1.0, 2007-10-25 Thursday).

ABL is a basic list library.

Copyright © 2004–2007 Anton Kulchitsky

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, with no Front-Cover Texts, and with no Back-Cover Texts. A copy of the license is not included in the manual itself. The manual must be distributed with the file containing the full text of the license though.

# Table of Contents

<b>1</b>	<b>Introduction</b> .....	<b>1</b>
<b>2</b>	<b>Installation</b> .....	<b>2</b>
<b>3</b>	<b>Description</b> .....	<b>3</b>
3.1	List Type .....	3
3.2	List Declaration .....	3
3.3	Interface .....	4
3.3.1	Initialization .....	4
3.3.2	Addition of Elements .....	4
3.3.3	Deletion of Elements .....	5
3.3.4	Point Movement .....	5
3.3.5	Getting Elements .....	6
3.3.6	List State .....	6
3.3.7	Reordering .....	6
3.3.8	Higher-Order Functions .....	6
3.4	The Loop Macro .....	7
3.5	Make and Destroy .....	7
<b>4</b>	<b>Examples</b> .....	<b>8</b>
4.1	List of Ints .....	8
4.2	Advanced List of Ints .....	8
4.3	List of Strings .....	10
4.4	Advanced List of Strings .....	10
<b>5</b>	<b>Planned Features</b> .....	<b>12</b>
<b>6</b>	<b>Change Log</b> .....	<b>13</b>
<b>7</b>	<b>Acknowledgements</b> .....	<b>14</b>
<b>Appendix A</b>	<b>Copying This Manual</b> .....	<b>15</b>
<b>Appendix B</b>	<b>Index</b> .....	<b>16</b>

# 1 Introduction

ABL is a flexible list library. It can be used to define linked lists of arbitrary type elements in C programs. It allows easily to define and effectively use complicated types like list of lists. ABL contains tools to make memory management easy. It also includes basic high order functions for searching, filtering, mapping, and effective reverse.

The library is implemented using (or abusing) C preprocessor rather than usual approach with void pointers. (Void pointers approach is described for example in “Mastering Algorithms with C” by Kyle Loudon.) The main difference between these approaches is that ABL is more designed as an extension to the language rather than a library. In our opinion, it allows more abstract definitions for a user and more flexible usage similar that is provided by template libraries in C++.

ABL also defines and was inspired by Lisp lists which maybe more convenient for Lisp/Scheme programmers to use.

The ABL can be included in any other libraries or programs that are compatible with LGPL by just copying a single header file.

ABL is not designed for huge lists that occupy a significant part of computer memory. ABL does not use any advanced garbage collection techniques neither it tries to keep allocated memory continues. However, the flexibility allows it to perform some tasks very effectively operating just on pointers.

Advantages of ABL are: flexibility, convenience, speciality, everything is in one header file (no linkage), and sometimes effectiveness. Finally, it is free software distributed under GNU LGPL condition.

*Note:* ABL is designed exclusively as a list library. It will remain a list library only. Different types of lists can be included in it later though.

## 2 Installation

To install the library in a standard location, use

```
./configure
make
su -c "make install"
```

In the case you installing it locally or to non-standard location, use

```
./configure --prefix=/desired/location
make
make install
```

instead. Do not forget to change `C_INCLUDE_PATH` to be able to reach the library.

You also may want just to use the code of the library without installing it. In this case, just copy file `'abl.h'` from `'src'` to your program files and use `#include "abl.h"` instead of `#include <abl.h>`.

## 3 Description

### 3.1 List Type

ABL defines an abstract linked list of elements of *uniform type*. (To use this list for storing elements of different types, one may use void pointers.) An element of the list consists of a pointer and a content. A pointer (*cdr*) points to the next element of the list. The content (*car*) contains the meaning of the element.

ABL list stores the pointer *head* to the first element of the list, the pointer *pointer* to some another element of the list, and the length of the list which is the number of elements in the list. It also may store pointers to functions make and destroy of car if necessary. See Section 3.5 [Make and Destroy], page 7. The *cdr* of the last element in the list is *nil*.

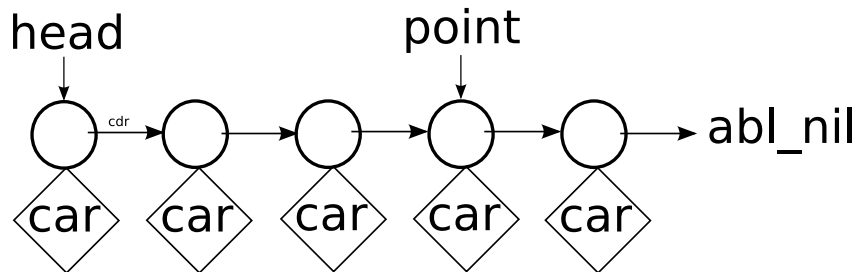


Figure 3.1: List of 5 elements. Point points to the fourth element in this example

### 3.2 List Declaration

To start to use the ABL in your program, all you need to do is to include ‘`abl.h`’ into your code:

```
#include <abl.h>
```

Then, you can start to declare your own list types with the macro `abl_typedef_list`.

`abl_typedef_list` *content\_type*, *new\_type\_name* [Macro]

The macro `abl_typedef_list` is used for a new list type definition. It must be used in the global name space outside of any function definition. It accepts two parameters. The first parameter is the type of the list element content (*car*). It can be a keyword or any type defined earlier by `typedef` or `#define` commands. It also can be a list type defined by the second parameter of `abl_typedef_list` command as well (see below).

The second parameter is your list type which is also a prefix to your list functions. It can be any C identifier. This macro introduces a new type (using `typedef`) for the list and a set of functions (interface) to work with this type.

For example, if you need a list of integers, you may define this type by the following command somewhere in the global name space of your program (outside of all function definitions):

```
abl_ttypedef_list( int, int_list )
```

You need to type this macro without `;` at the end. Now you can declare variables of your list type (if you are from C++ world, you may think about them as objects of this list class):

```
int_list a,b,c;
```

for our example of `int_list`. You also can declare pointers to this type and then use `malloc` for allocation of memory for this type and, of course, `sizeof` function to determine the size of the type:

```
int_list* pa = (int_list*) malloc( sizeof(int_list) );
```

You can produce types using new declared list type. For example, you may type

```
abl_ttypedef_list( int_list, intlists_list )
```

to determine the list of lists of integers.

### 3.3 Interface

Defining the type, you define the whole set of functions to work with it. Every function has a prefix which is your list type followed by the underscore. The following description is defined without that prefix. Thus, to call `init` on your list of integers defined in the previous section, you need to call actually `int_list_init` for it. The first parameter of any interface function is a pointer to a variable of this list type.

#### 3.3.1 Initialization

These functions must be used before any other functions are applied. Function `init` must be called before list usage. Function `set_md` is optional.

```
void init list_type* plist [Function]  

  init initializes the list pointed by plist before any usage. This function must be called before of any other operation on the particular variable of your list type. This function set make and destroy to NULL. See Section 3.5 \[Make and Destroy\], page 7.
```

```
void set_md list_type* plist, int (*make)(content_type*,void*), void [Function]  

  (*destroy)(type_of_car*)  

  set_md set make and destroy functions for the list content. Either make or destroy (or both) can be set to NULL. In this case, these make or destroy will be ignored. See Section 3.5 \[Make and Destroy\], page 7.
```

#### 3.3.2 Addition of Elements

```
int add_head list_type* plist, content_type* pcar [Function]  

  Adds a head and copies the content from pcar to element's content. It returns ABL_ERR_ALLOC if there was a memory allocation error or ABL_OK on success.
```

**int create\_head\_c** *list\_type\* plist, content\_type\* pcar, void\* cparam* [Function]  
 Creates a new head of the list. The previous head becomes the second element. It does not initialize the added head content explicitly. However, the element can be initialized by `make` function. If `make` is set by `set_md` function, `create_head_c` calls the `make` with `cparam` as a parameter on a new allocated car (content). See [Section 3.5 \[Make and Destroy\], page 7](#). It returns `ABL_ERR_ALLOC` if there was a memory allocation error, any error code returned by the `make`, or `ABL_OK` on success. You can set `cparam` to `NULL` if you do not use `make`. *Note:* `make` can be used to initialize the element. See [Chapter 4 \[Examples\], page 8](#).

### 3.3.3 Deletion of Elements

**void clean** *list\_type\* plist* [Function]  
 Removes all elements from the list. It calls `del_head` until the list is empty. This function must be called at least once after all work on the list unless it is absolutely clear that the list is empty.

**void del\_head** *list\_type\* plist* [Function]  
 Deletes the head element from the list and move the head to the next element. It releases the memory from the head content and from the head. It also calls the `destroy` on the car if the `destroy` is not `NULL`. See [Section 3.5 \[Make and Destroy\], page 7](#). Programmers that do not use the `destroy`, must care about memory freeing if the content contains pointers. `del_head` does nothing for the empty list.

**int del\_after\_point** *list\_type\* list\_type\* plist* [Function]  
 Deletes the element next after the point from the list. It releases the memory from the element content and from the element. It also calls the `destroy` on the car if the `destroy` is not `NULL`. See [Section 3.5 \[Make and Destroy\], page 7](#). Programmers that do not use `destroy` function must care about memory freeing if content contains pointers.

If the pointer or the next element after the pointer is `abl_nil`, nothing is done and error code is returned.

Error codes:

- `ABL_OK` if the element was deleted successfully
- `ABL_ERR_NIL` if pointer points to nil or list is empty
- `ABL_ERR_DELNIL` if the next element is `NIL`

### 3.3.4 Point Movement

**void move\_point\_head** *list\_type\* plist* [Function]  
 Moves the point to the head.

**void move\_point\_next** *list\_type\* plist* [Function]  
 Moves the point to the next element. If point points to *nil*, it does nothing.

### 3.3.5 Getting Elements

- content\_type\*** `get_head` *list\_type\* plist* [Function]  
 same as `get_car`, `get_head` returns the car (pointer to the content) of the head element.
- content\_type\*** `get_car` *list\_type\* plist* [Function]  
 same as `get_head`, `get_car` returns the pointer to the content of the head element.
- content\_type\*** `get_point` *list\_type\* plist* [Function]  
`get_point` returns the car (pointer to the content) of the element pointed by the point. It returns NULL if the pointer points to `abl_nil`.
- content\_type\*** `get_point_next` *list\_type\* plist* [Function]  
 Returns the pointer to the content of the element next after the list pointer or NULL if there is no such an element.

### 3.3.6 List State

These functions return the state of the list: if it is empty, where the point is, the length of the list.

- int** `is_empty` *list\_type\* plist* [Function]  
 Returns 1 if the list is empty and 0 otherwise.
- int** `is_point_nil` *list\_type\* plist* [Function]  
 Returns 1 if the point points to the *nil* that means to the end of the list.
- size\_t** `length` *list\_type\* plist* [Function]  
 Returns the number of elements in the list. Operation order is O(1) due to list stores the length in the internal variable.

### 3.3.7 Reordering

- void** `nreverse` *list\_type\* plist* [Function]  
 Reverse (destructively) the list in order. The operation is of O(n) order (performs 4 assignments per element).

### 3.3.8 Higher-Order Functions

This class of functions contains operators on the list that apply some functions to the elements or content and modify the list or return some values.

- int** `deleteif` *list\_type\* plist, int (\*filter\_cond)(content\_type\*)* [Function]  
 Deletes all elements in the list that satisfy the *filter\_cond*. The function `filter_cond` must return 0 if the element is *not* to be deleted. `deleteif` returns the error code (ABL\_OK means no errors).
- content\_type\*** `findif` *list\_type\* plist, int (\*predicate)(content\_type\*)* [Function]  
 Returns pointer to the element content (car) of the list that first satisfies of the predicate condition.
- void** `mapcar` *list\_type\* plist, void (\*f)(content\_type\*)* [Function]  
 Applies `f` to every element in the list.

### 3.4 The Loop Macro

The loop macro is defined to simplify routine looping code.

**ABL\_BEGIN\_LOOP** *name\_of\_list\_type lst* [loop on abl]

Starts the loop for every element of the list *lst*. The reference to the element is performed by `get_point` function. Two parameters are necessary: the list type and the pointer to the list. See [Chapter 4 \[Examples\], page 8](#).

**ABL\_END\_LOOP** *name\_of\_list\_type lst* [loop on abl]

Ending of the loop facility. See `ABL_BEGIN_LOOP` for details.

### 3.5 Make and Destroy

Make and destroy feature intend to improve the memory management of the list. They are created to automate memory allocation and deallocation for list of pointers of list of elements that contains some pointers. The simplest example can be a list of `char*`, see examples below. Make also proved to be useful beyond this narrow application.

Destroy is a function that returns void. It has only one parameter, a pointer to the car type. Thus, if the list was declared

```
abl_typedef_list( char*, str_list )
```

then the destructor must be declared

```
void mydestr( char** );
```

(`mydestr` can be replaced by any desired name.) Then the destroy can be set for the list by `str_list_set_md` function.

If `destroy` is different from `NULL`, it is called by any deletion function on the car of the element to delete. See [Section 3.3.3 \[Deletion of Elements\], page 5](#). Usually, `destroy` contains only calls of `free` function. Thus, for the example above, we would have

```
void mydestr( char** pstr )
{
    free( *pstr );
}
```

The `make` is a function that returns int, the error code. It must return `ABL_OK` on success. Otherwise, it can return any error code but 0. This error code will be returned by some addition functions that call the `make` (namely `create_head`). The constructor has two parameters: (1) The pointer to the car of the element to construct and (2) The void pointer to some initialization value. This value is used to determine what kind of initialization is to be done.

The constructor is called by `create_head` if and only if the constructor set to any other value than `NULL`. The last parameter of these functions is sent to the `make`.

## 4 Examples

### 4.1 List of Ints

The program creates a list of integers, initializes it with 0,1,2,3,4 in a loop, and prints all elements from the head to the tail. It uses the simplest functions and the LOOP macro for printing elements.

```
#include <stdio.h>
#include <stdlib.h>
#include <abl.h>

/* declare the int_list type */
abl_typedef_list( int, int_list )

int main()
{
    int_list lst;          /* list of ints */
    int i;                /* iterator */

    int_list_init( &lst ); /* initialize the list */

    /* initialize list elements by meaning of loop index */
    for( i=0; i<5; ++i ) int_list_add_head( &lst, &i );

    /* print all elements from the list */
    ABL_BEGIN_LOOP( int_list, &lst ) {
        printf( "%d\n", *int_list_get_point(&lst) );
    } ABL_END_LOOP( int_list, &lst );

    /* clean the list */
    int_list_clean( &lst );

    return 0;
}
```

### 4.2 Advanced List of Ints

This example demonstrates more advanced usage of list.

The program creates a list of integers, initializes it with 0,1,2,3,4,5,6,7,8,9,10 in a loop, finds and prints the first element that divides 4 (it is 8 of course), deletes all odd elements from the list, reverse the list, and prints all elements from the head to the tail using `mapcar` function.

```
#include <stdio.h>
#include <stdlib.h>
#include <abl.h>
```

```
/* declare the int_list type */
abl_typedef_list( int, int_list )

/* print an elements */
void printint( int* a )
{
    printf( "%d\n", *a );
}

int isodd( int* a )
{
    return ( *a % 2 == 1 );
}

int isdiv4( int *a )
{
    return ( *a % 4 == 0 );
}

int main()
{
    int_list lst;          /* list of ints */
    int i;                /* iterator */

    int_list_init( &lst );    /* initialize the list */

    /* initialize list elements by meaning of loop index */
    for( i=0; i<=11; ++i ) int_list_add_head( &lst, &i );

    /* find the first element that %4==0, and print it */
    printf( "The first element that divides 4 is [%d]\n",
           *int_list_findif( &lst, isdiv4 ) );

    /* delete all odd elements */
    int_list_deleteif( &lst, isodd );

    /* reverse the list to make a head == 0 */
    int_list_nreverse( &lst );

    /* print all elements from the list */
    int_list_mapcar( &lst, printint );

    /* clean the list */
    int_list_clean( &lst );

    return 0;
}
```

### 4.3 List of Strings

This program creates a list of strings, add two elements "First String" and "Second String" to it, print the list content and then cleans the list properly.

This example is created to demonstrate how to allocate memory for the elements without make/destroy usage.

```
#include <stdio.h>
#include <stdlib.h>
#include <abl.h>

abl_typedef_list( char*, str_list )

int main()
{
    str_list lst;
    char* cc;

    /* List initialization */
    str_list_init( &lst );

    /* Set two list elements "First String" and "Second String" */
    str_list_create_head( &lst, NULL );
    *str_list_get_head( &lst ) = strdup("First String");

    str_list_create_head( &lst, NULL );
    *str_list_get_head( &lst ) = strdup("Second String");

    /* Print the list */
    ABL_BEGIN_LOOP( str_list, &lst ) {
        printf( "%s\n", *str_list_get_point( &lst ) );
    } ABL_END_LOOP( str_list, &lst );

    /* Free the memory from each string */
    ABL_BEGIN_LOOP( str_list, &lst ) {
        free( *str_list_get_point( &lst ) );
    } ABL_END_LOOP( str_list, &lst );

    /* Delete all elements */
    str_list_clean( &lst );

    return 0;
}
```

### 4.4 Advanced List of Strings

This program creates a list of strings, add two elements "First String" and "Second String" to it, print the list content with mapcar, and then cleans the list properly.

This test is created to demonstrate how to allocate and initialize memory for the elements WITH make/destroy usage.

```
#include <stdio.h>
#include <stdlib.h>
#include <abl.h>

abl_typedef_list( char*, str_list )

void printstr( char** pstr )
{
    printf( "%s\n", *pstr );
}

int str_list_make( char** pstr, void* inistr )
{
    *pstr = strdup( (char*)inistr );
    if( !(*pstr) ) return ABL_ERR_ALLOC;
    return ABL_OK;
}

void str_list_destroy( char** pstr )
{
    free( *pstr );
}

int main()
{
    str_list lst;

    /* List initialization */
    str_list_init( &lst );
    str_list_set_md( &lst, str_list_make, str_list_destroy );

    /* Set two list elements "First String" and "Second String" */
    str_list_create_head( &lst, "First String" );
    str_list_create_head( &lst, "Second String" );

    /* Print the list */
    str_list_mapcar( &lst, printstr );

    /* Delete all elements */
    str_list_clean( &lst );

    return 0;
}
```

## 5 Planned Features

This section is a short list of planned features with explanations.

1. Add other high-order functions: like `foldr/foldl` (reduce?) and `sort`.
2. Consider usefulness and addition of functions like `add_after_point` and `add_after_point_c`.
3. Adding functions for retrieving/copying sublists (like `cdr`).

## 6 Change Log

### New in version 1.0 compare to version 0.99

This is a major release with a lot of big changes.

1. `car` is no more a pointer (after discussion on <http://linux.org.ru>). This changes how list elements are stored in memory.
2. Function `add_head_c` is deleted! See [Section 3.3 \[Interface\]](#), page 4.
3. Function `create_head_c` is deleted. Function `create_head` was modify to unify both creation methods. See [Section 3.3 \[Interface\]](#), page 4.
4. `Make`/destructor are renamed to `make`/destroy. See [Section 3.5 \[Make and Destroy\]](#), page 7.
5. Major revision of the documentation. All examples and tests are redone.
6. ABL logo is created. ABL is now not “Atoku’s Basic List” but rather “Abstract Basic List”. Ha-ha!

### New in version 0.99 compare to version 0.98

This is a major release in spite of the minor change in numbers. After extensive testing, it supposed to be version 1.0.

1. New functions: `mapcar`, `add_head_c`, `create_head`, `create_head_c`. See [Section 3.3 \[Interface\]](#), page 4.
2. Constructor/Destructor feature. Well, it is pretty advances thing. I did not see anything similar before. See [Section 3.5 \[Make and Destroy\]](#), page 7.
3. Completely revised documentation. There are some examples added to the doc as well.

### New in version 0.98 compare to version 0.9/0.95

1. New functions: `del_after_point` and `get_point_next`. See [Section 3.3 \[Interface\]](#), page 4.
2. New macros: `ABL_BEGIN_LOOP` and `ABL_END_LOOP`. See [Section 3.4 \[The Loop Macro\]](#), page 7.
3. Doxygen compatible comments. `abl.h` can be used to generate doxygen documentation for your program (Documentation of this feature is to be added later).
4. `point_next` changed the name to `move_point_next`.

## 7 Acknowledgements

Thanks *k\_andy* (Andrei from Donetsk) from <http://linux.org.ru> for discovering a design bug and advising how to fix it in transition from version 0.99 to version 1.0.

## Appendix A Copying This Manual

This documentation is free.

I like the GNU FDL, however, it seems for me too large to include into the documentation. Moreover, I do not like all examples in the text to be dual-licensed. Thus, this documentation is distributed under the GNU GPLv3 license. See Free Software Foundation web site for details on it.

## Appendix B Index

### A

ABL_BEGIN_LOOP on abl .....	7
ABL_END_LOOP on abl .....	7
abl_typedef_list .....	3
add_head .....	4
addition of elements .....	4

### C

car .....	3
cdr .....	3
change log .....	13
clean .....	5
create_head_c .....	5

### D

declaration .....	3
del_after_point .....	5
del_head .....	5
deleteif .....	6
deletion of elements .....	5
description .....	3
destroy .....	4, 7

### E

examples .....	8
----------------	---

### F

findif .....	6
--------------	---

### G

get_car .....	6
get_head .....	6
get_point .....	6
get_point_next .....	6
getting elements .....	6

### H

head .....	3
higher-order Functions .....	6

### I

init .....	4
initialization .....	4
installation .....	2
interface .....	4
is_empty .....	6
is_point_nil .....	6

### L

length .....	3
length .....	6
list .....	3
list element .....	3
list state .....	6
loop macro .....	7

### M

make .....	4, 7
mapcar .....	6
move_point_head .....	5
move_point_next .....	5

### N

nil .....	3
nreverse .....	6

### P

planned features .....	12
point movement .....	5
pointer .....	3

### R

reordering .....	6
------------------	---

### S

set_md .....	4
--------------	---