Fundamentals of Programming 2

Singly Linked Linear List and Recursion

Arkadiusz Chrobot

Department of Computer Science

March 30, 2020

Outline

Introduction

As it was stated in previous lectures, the base type of such data structures as stacks and queues is recursive. The type is based on a structure that contains a pointer field which can point to other variables of the same type. Those variables are elements (in other words: nodes) of a data structure. This description also fits to the singly linked linear list, which is a more general data structure than the two aforementioned and thus it requires more of basic operations to be implemented. A question arises, if implementing them with the use of recursive functions would be beneficial? It shows up, that applying the recursive approach simplifies many aspects of list handling. To prove it the program from the previous lecture has been modified to use mostly recursive functions. Also the relations between the recursion and the functional programming paradigm is explained and the usage of such a model in the C language is demonstrated.

Assumptions

The program uses a singly linked linear list to store natural numbers in the ascending order. So, the list is sorted. Although only natural numbers are stored in the list by the program, the list can store any number that is of the `int` type.

Base Type and List Pointer

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

struct list_node {
    int data;
    struct list_node *next;
} *list_pointer;
```

Base Type and List Pointer

The beginning of the program is unchanged. The same header files are included as before. The definition of the base type and the declaration of the list pointer also stay the same. The assumption about the list pointer discussed in the previous lecture also holds — the pointer always should point to a fist element of the list or have the `NULL` value.

Operation of Adding an Element to the List

It can be concluded, by applying the recursive approach to the operation of adding a new element to the singly linked linear list, that there are two cases to consider:

1. a new element is added to an empty (nonexistent) list,
2. a new element is added to an existing (nonempty) list.

The second case covers all situations related to inserting a new element to the existing sorted list, i.e. adding at the beginning, inside and at the end. Please note, that there is no need for the operation of creating the list in the recursive approach. It is covered by the first case.
The operation of adding a new element to the list is implemented with the use of a "main" function and a helper function. The latter is described first.

Operation of Adding an Element to the List

The `create_and_add_node()` Function

```
1  int create_and_add_node(struct list_node **list_pointer,
2                          int number)
3  {
4      struct list_node *new_node = (struct list_node *)
5                          malloc(sizeof(struct list_node));
6      if(!new_node)
7          return -1;
8      new_node->data = number;
9      new_node->next = *list_pointer;
10     *list_pointer = new_node;
11     return 0;
12 }
```

Operation of Adding an Element to the List

The `create_and_add_node()` Function

The function, as its name suggests, creates a new element and inserts it to the list. It returns a value of the `int` type, which is a status of the operation. It also has two parameters. The first one is a pointer to a pointer of the list base type. The second one is a variable of the `int` type and it is used for passing a number which should be stored in the new element. Please note, that the function body is quite simple. In the lines no. 4 and 5 the function tries to allocate memory for the new element. If the allocation fails the function returns `-1` and exits. If it is however successful, then the number is assigned to the element (line no. 8) and in the `next` field of the same element is stored an address stored in a variable pointed by the pointer to a pointer (line no. 9). Next, the address of the new element is assigned to the aforementioned variable, and the function exits returning zero.

Operation of Adding an Element to the List

The `create_and_add_node()` Function

The `create_and_add_node()` function performs activities which correspond to all cases of adding a new element to the sorted list in the program from the previous lecture and additionally to the operation of creating a new list (adding the first element to the list). A question arises, how such a simple function can cover all those cases that in the previous version of the program required defining several separate functions? The answer is given after the `add_node()` function, which is responsible for finding a spot in the list where the new element should be added and for invoking the `create_and_add_node()` function, is analysed.

Operation of Adding an Element to the List

The `add_node()` Function

```
1  int add_node(struct list_node **list_pointer, int number)
2  {
3      if(*list_pointer!=NULL && (*list_pointer)->data<number)
4          return add_node(&(*list_pointer)->next,number);
5      else
6          return create_and_add_node(list_pointer,number);
7  }
```

Operation of Adding an Element to the List

The `add_node()` Function

The `add_node()` function has the same, with the exception of the name, prototype as the `create_and_add_node()` function. Also the meaning of its return value is the same as in the previously described function. However, the pointer to a pointer parameter has a different role. When the function is invoked for the first time the parameter stores the address of the list pointer, so any operation that changes the value of the parameter is also changing the value of the pointer. If the function is invoked recursively (line no. 4) then the parameter will store the address of the `next` field of the list element which was pointed by the parameter previously. Let's analyse all possible scenarios of the function behaviour:

Adding an element to an empty list.

The `*list_pointer!=NULL` expression, which is a part of the condition in the 3rd line, is false, thus the `create_and_add_node()` function is called immediately in the line no. 6, and it creates and adds the first and only element to the list.

Operation of Adding an Element to the List

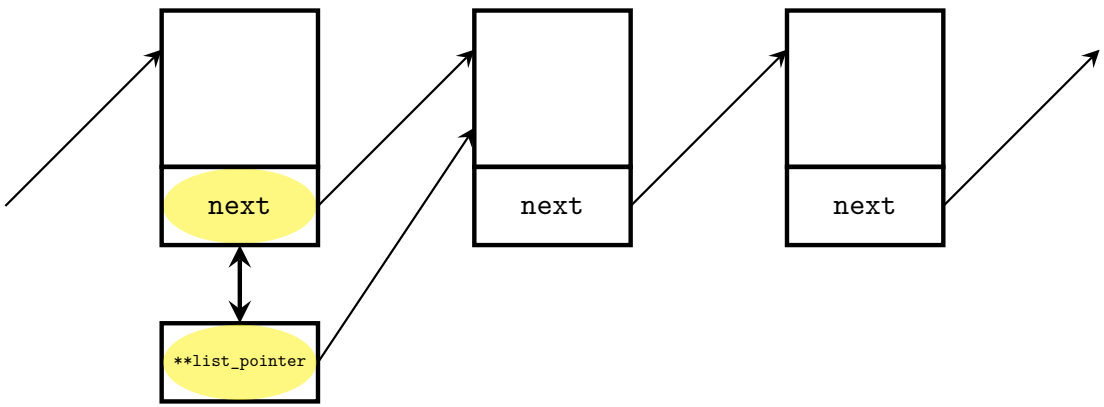
The `add_node()` Function

Adding an element at the beginning of the list.
In this case the first expression in the condition in 3rd line is true, but the second one (the one after the `&&` operator) is false, thus once again the `create_and_add_node()` function is called immediately in the 6th line, which adds an element at the beginning of the list.

Adding an element inside the list.
This time both expressions in the condition in 3rd line are true and the function is invoked recursively (line no. 4). Those invocations are preformed as long as the second expression in the aforementioned condition becomes false. When it happens it means that the instance (invocation) of the function, for which it happened, should add a new element before the element pointed by the variable, which address is stored in the `list_pointer` parameter. The variable is the `next` field of the element that precedes in the list the element before which the new element has to be inserted. The case is illustrated in the next slide.

Operation of Adding an Element to the List

Relationship Between `struct list_node **list_pointer` and the `next` Field



Explanation of the relationship between pointer to a pointer and the `next` field of a list element

Operation of Adding an Element to the List

Relationship Between `struct list_node **list_pointer` and the `next` Field — Comment

The `next` field and the `**list_pointer` pointer are marked in the figure in the previous slide by yellow ellipses. It means that those two variables should be considered as one, i.e. making changes to one of them will cause immediate change of the value of the second one.

Operation of Adding an Element to the List

The `add_node()` Function

Adding an element inside the list — continued.
The element is created and added by the `create_and_add_node()` function invoked in the 6th line.

Adding at the end of the list.
In this case, after a sequence of recursive invocations, an instance of the function is created, for which the first expression in the condition in the 3rd line is not satisfied. It means that the new element should be added at the end of the list. As in the previous cases the element is actually created and added by the `create_and_add_node()` function invoked in the 6th line of the `add_node()` function.

Operation of Removing an Element From the List

The recursive version of the operation of removing an element from a sorted singly linked linear list consist just of locating an element storing the given value in the list, and removing it. The course of the operation is always the same, regardless of where the element is in the list. Let's remind, that if there is more then one element in the list that has the given value, then removing the one that is found as the first satisfies the assumptions for the operation. The next slide contains the definition of a recursive function that implements the operation of removing a single element from the list. Its source code is so short that it doesn't require partitioning into separate subroutines.

Operation of Removing an Element From the List

The `delete_node()` Function

```
1  void delete_node(struct list_node **list_pointer, int number)
2  {
3      if(*list_pointer) {
4          if((*list_pointer)->data == number) {
5              struct list_node *next = (*list_pointer)->next;
6              free(*list_pointer);
7              *list_pointer = next;
8          } else
9              delete_node(&(*list_pointer)->next, number);
10     }
11 }
```

Operation of Removing an Element From the List

The `delete_node()` Function

The function presented in the previous slide doesn't return any value, since it doesn't generate any exceptions and its effects are visible after the list is printed on the screen. The function has two parameters — the first one is a pointer to a pointer of the `struct list_node` type, and the second one is of the `int` type. By the latter the number is passed that should be stored in the list element to be removed. When the function is called for the first time it checks if the list is not empty (line no. 3). If the list exists then the function checks if the first element of the list contains the value passed by the second parameter. If so, then the element should be removed. It that case the address stored in the `next` field of the element is assigned to a local pointer (line no. 5) and the memory allocated for the element is freed. Next, the `*list_pointer` variable is assigned an address which was stored in the `next` field of the removed element. It is an address of then second and now first element of the list.

Operation of Removing an Element From the List

The `delete_node()` Function

If however the condition in the 4th line is not satisfied for the first element, then the function is invoked recursively (line no. 9) in order to find such an element of the list that would satisfy it. Please note, that as the first argument for the call is passed the address of the `next` field of the list element accessed in the current instance of the function. Thus any modification of the value of the first parameter made in the next instance of the function will be also made to the value of the field. In that way the statements in lines no. 5, 6 and 7 handle also the cases where the element is removed from the inside of the list and at its end. If the list doesn't contain an element to be removed, then the function is eventually called recursively for the `next` field that contains the `NULL` value. In that case it does nothing, just exits, like its previous recursive invocations. That way the case of removing a nonexistent element from the list is handled by the function.

Operation of Printing the Content of the List

The recursive version of the operation of printing the content of the list is as short as its iterative version. It can be simply described in the following way:

1. if the list exists then print the value of its first element,
2. print the content of the rest of the list.

The next slide contains definition of a function that implements the operation.

Operation of Printing the Content of the List

The `print_list()` Function

```
1  void print_list(struct list_node *list_pointer)
2  {
3      if(list_pointer) {
4          printf("%d ",list_pointer->data);
5          print_list(list_pointer->next);
6      } else
7          puts("");
8  }
```

Operation of Printing the List

The `print_list()` Function

The `print_list()` function doesn't return any value — its results are visible on the screen. It takes only one argument, and it is the list pointer. In the 3rd line the function checks if the pointer passed by the function's parameter is not empty. If the condition is satisfied then the function prints the value of the `data` field of the element of the list which address is passed by the parameter and invokes itself recursively taking as an argument the address stored in the `next` field of the currently accessed element of the list. If the address in not `NULL` then another element of the list exists for which the next instance of the function will perform the statements in the 4th and 5th lines. Otherwise the next recursive invocation of the function will move the cursor to the next line on the screen, using the `puts()` function, and exit. In that case also the previous recursive invocations of the `print_list()` function will exit.

Operation of Printing the List in Reversed Order

The `print_list_inversely()` Function

It occurs that a small modification of the `print_list()` function makes it possible to perform an operation which was very hard to do in the iterative version — printing the values of the elements of the list in the reversed order. It suffice to swap the statements in the 3rd and 4th lines, so that the value of the element is printed *after* the function returns from the recursive invocation. This version of the `print_list()` function is not calling the `puts()` function. The latter should be called after the value of the first element is displayed on the screen and it is difficult to detect. The cursor can be moved to the next line on the screen after the function exits. The next slide contains the definition of the `print_list_inversely()` function which contains the described changes.

Operation of Printing the List in Reversed Order

The `print_list_inversely()` Function

```
1  void print_list_inversely(struct list_node *list_pointer)
2  {
3      if(list_pointer) {
4          print_list_inversely(list_pointer->next);
5          printf("%d ",list_pointer->data);
6      }
7  }
```

Operation of Removing the List

The operation of removing of all the element of the list is similar to the operation of printing the content of the list in the reversed order. The difference is in the declaration of the parameter of the function (this time it is a pointer to a pointer) and in the operation that is performed on the element. The next slide contains the definition of the `remove_list()` function which performs such an operation.

Operation of Removing the List

The `remove_list()` Function

```
1  void remove_list(struct list_node **list_pointer)
2  {
3      if(*list_pointer) {
4          remove_list(&(*list_pointer)->next);
5          free(*list_pointer);
6          *list_pointer = NULL;
7      }
8  }
```

Operation of Removing the List

The `remove_list()` Operation

The function doesn't return any value, but has a single parameter which is a pointer to a pointer of the `struct list_node` type. When the function is invoked for the first time it checks (line no. 3) if the list exists. If so, it calls itself recursively in the 4th line. It keeps invoking itself until one of its instances is called for the `next` field of the last element in the list. The filed stores the `NULL` value and the instance of the function does nothing except exiting. The control flow goes back to 5th line of the instance of the function called for the last element of the list. Here, the instance frees the memory allocated for that element and assigns the `NULL` value to the `next` filed of the element that was last but one on the list. The instance exits and the control flow returns to the instance called for the last but one element, which will repeat the described activities for that element. The returns will be finished when the first element of the list is deallocated and the instance invoked for that element will exit.

Operation of Removing the List

The `remove_list()` Function

Please note, that the 4th and 5th lines of the described function cannot be swapped, otherwise the function would be invoked recursively for nonexistent elements of the list.

Functional Approach

The recursion is strongly related to the functional paradigm of programming, where it is used in place of the iteration. It is not the only one element of this paradigm. In the functional model the most important concept is the function. The variables are immutable, i.e. the values that are assigned to them cannot be changed. Thus any subroutine that operates on such variables has no side-effects. Functions can be passed by parameters to other functions which perform operations on them or use them for performing other operations. The latter functions are called the *higher order functions.* The C language doesn't support directly the higher order functions or the functional paradigm, but similar effects can be achieved with the use of function pointers.

Performing Operations on the List that Return No Results

In the next slide a function is defined that recursively traverses the list and performs an operation for each of its elements. The operations is defined by another function, which address is passed to as an argument to the former function.

Performing Operations on the List Returning No Results

The `iterate_list()` Function

```
1  void iterate_list(struct list_node *list_pointer,
2                            void (*action)(struct list_node *))
3  {
4      if(list_pointer) {
5          if(action)
6              action(list_pointer);
7          iterate_list(list_pointer->next,action);
8      }
9  }
```

Performing Operations on the List Returning No Results

The `iterate_list()` Function

The function doesn't return any value but has two parameters. The first one is used for passing the list pointer and the second for passing the address of a function which also doesn't return any value but takes as an argument the address of an element of the list for which it performs an operation. The `iterate_list()` function is a higher order function. In the 4th line it checks if the list pointer which it received by its first parameter is not empty. If the condition is satisfied the function checks if the pointer to the function is also not empty. If not, no operation is performed on the currently accessed list's element, just the `iterate_list()` calls itself recursively (line no. 7). Otherwise the function pointed by the `action` pointer is invoked for the element of the list that is accessed by the current instance of the `iterate_list()` function.

Performing Operations on the List Returning No Results

The `print_element()` Function

```
1  void print_element(struct list_node *list_pointer)
2  {
3      printf("%d ",list_pointer->data);
4  }
```

Performing Operations on the List Returning No Results

The `print_element()` Function

The `print_element()` function defines an example operation for a single element of the list — it prints the value of the `data` field of the element on the screen. If the `iterate_list()` function is called with the address of the `print_element()` function as its second argument, then it will print the values of all elements of the list on the screen. The only difference between the result of the `iterate_list()` function and the result of the `print_list()` function is that the former doesn't moves the cursor to the next line of the screen.

Performing Operations on the List Returning No Results

The `double_element_value()` Function

```
1  void double_element_value(struct list_node *list_pointer)
2  {
3      list_pointer->data*=2;
4  }
```

Performing Operations on the List Returning No Results

The `double_element_value()` Function

The function, when invoked by the `iterate_list()` function, doubles the value of the `data` field of the element which address is passed to it by its parameter. Hence, in that case the `iterate_list()` function doubles values of all elements of the list.

Performing Operations on the List That Return Results

The presented higher order function makes it possible to perform operations which change the values of the elements of the list, like the `double_element_value()` function. It is not however completely compatible with the functional paradigm of programming in which once assigned variables don't change their values. Let's try to define another higher order function which returns a result of the operations performed on the list by functions invoked by it. The definition of the function is given in the next slide.

Performing Operations on the List That Return Results

The `iterate_list_with_result()` Function

```
1  int iterate_list_with_result(struct list_node *list_pointer,
2          int (*action)(int result, struct list_node *list_pointer))
3  {
4      int result=0;
5      for(; list_pointer; list_pointer=list_pointer->next)
6          if(action)
7              result=action(result,list_pointer);
8      return result;
9  }
```

Performing Operations on the List That Return Results

The `iterate_list_with_result()` Function

The function is similar to the `iterate_list()` function, but in contrast to it the former function takes as a second argument an address of a function that returns a value of the `int` type and has two parameters. By the first parameter a variable is passed, which is used for accumulating results of previous operations carried out by the function on elements of the list. Such a parameter is redundant in languages which directly support the functional programming model, but it is necessary in the C language. The second parameter is a pointer to an element of the list for which the operation has to be carried out. The `iterate_list_with_result()` function uses the iteration instead of the recursion (line no. 5) to traverse the list. The loop allows it to use the local variable `result` in the 7th line for storing the results of previously carried operations on the already visited elements of the list. To make it possible, the variable is passed as the first argument to the function invoked with the help of the `action` parameter.

Performing Operations on the List That Return Results

The `add_up()` Function

```
1  int add_up(int result, struct list_node *list_pointer)
2  {
3      return result+list_pointer->data;
4  }
```

Performing Operations on the List That Return Results

The `add_up()` Function

The function presented in the previous slide is an example of a function that can be invoked by the `iterate_list_with_result()` function. If it happens the latter will return the sum of the values of all elements of the list. The value will be correct if it is in the range of the `int` type.

Performing Operations on the List That Return Results

The `count_elements()` Function

```
1  int count_elements(int result, struct list_node *list_pointer)
2  {
3      return result+1;
4  }
```

Performing Operations on the List That Return Results

The `count_elements()` Function

If the function presented in the previous slide is invoked by the `iterate_list_with_result()` function then the latter will return the number of the elements in the list. The result will be correct if the number is in the range of the `int` type.

Performing Operations on the List That Return Results

Summary

Please note, that the presented solution limits the possible operations to only those that return as a result a value of the `int` or compatible type. The typical functional programming languages doesn't have such limitations, because they are in most cases dynamically typed. It means that the programmer doesn't have to define the types for variables, parameters and the values returned by the functions. They are dynamically defined when the program runs.

The `main()` Function

In the main function of the program all the previously defined functions are invoked. Just as in the program presented in the previous lecture, their behaviour is tested for all the most important cases.

The main() Function

First Part

```
1  int main(void)
2  {
3   int i;
4   for(i=1; i<5; i++)
5    if(add_node(&list_pointer,i)==-1)
6      fprintf(stderr,"Adding an element to the list exception!\n");
7   for(i=6; i<10; i++)
8    if(add_node(&list_pointer,i)==-1)
9      fprintf(stderr,"Adding an element to the list exception!\n");
10  print_list(list_pointer);
```

The `main()` Function

First Part

In the first part of the `main()` function, which is presented in the previous slide, a list is created that contains natural numbers ranging for `1` to `4` and form `6` to `9`. Please note, that the whole operation is carried out with the use of the `add_node()` function. Each time the function is called its result is checked. Should it be equal `-1` the program prints a message about the failure of adding a new element to the list. Please observe, that the first argument of the `add_node()` function is the address of the list pointer. After the list is created its content is displayed on the screen by the `print_list()` function.

The main() Function

Second Part

```
1   if(add_node(&list_pointer,0)==-1)
2     fprintf(stderr,"Adding an element to the list exception!\n");
3   print_list(list_pointer);
4   if(add_node(&list_pointer,5)==-1)
5     fprintf(stderr,"Adding an element to the list exception!\n");
6   print_list(list_pointer);
7   if(add_node(&list_pointer,7)==-1)
8     fprintf(stderr,"Adding an element to the list exception!\n");
9   print_list(list_pointer);
10  if(add_node(&list_pointer,10)==-1)
11    fprintf(stderr,"Adding an element to the list exception!\n");
12  print_list(list_pointer);
```

The `main()` Function

Second Part

In the second part of the `main()` function, single elements are added at the beginning, in the middle and at the end of an existing list. Also an element is added that stores a value that is already in the list. After each such an operation is carried out the result returned by the `add_node()` function is checked and the content of the list is displayed on the screen with the use of the `print_list()` function.

The main() Function

Third Part

```
1   print_list_inversely(list_pointer);
2   puts("");
3   delete_node(&list_pointer,0);
4   print_list(list_pointer);
5   delete_node(&list_pointer,1);
6   print_list(list_pointer);
7   delete_node(&list_pointer,1);
8   print_list(list_pointer);
9   delete_node(&list_pointer,4);
10  print_list(list_pointer);
11  delete_node(&list_pointer,7);
12  print_list(list_pointer);
13  delete_node(&list_pointer,10);
14  print_list(list_pointer);
```

The `main()` Function

Third Part

In the third part of the `main()` function the list content is printed in the reversed order with the use of the `print_list_inversely()` function. The cursor is moved to the next line on the screen after the function exits, more precisely in the 2nd line of the described part of the program. Next, elements from the beginning, the middle and the end of the list are removed. The program also tries to remove a nonexistent element (line no. 7) and an element that stores a number which is represented twice in the list (line no. 11). After each of the operations is performed, the content of the list is displayed on the screen.

The main() Function

Forth Part

```
1   iterate_list(list_pointer,NULL);
2   iterate_list(list_pointer,print_element);
3   puts("");
4   iterate_list(list_pointer,double_element_value);
5   iterate_list(list_pointer,print_element);
6   puts("");
7   printf("The sum of the values of the list's elements: %d\n",
8                 iterate_list_with_result(list_pointer,add_up));
9   printf("The number of the list's elements: %d\n",
10        iterate_list_with_result(list_pointer,count_elements));
11  remove_list(&list_pointer);
12  return 0;
13 }
```

The `main()` Function

The Forth Part

In the forth part of the `main()` function are invoked all functions that in the functional programming paradigm are called higher order functions. First the `iterate_list()` function is called, but with no valid address of a function that performs an operation on the list. This verifies if the function behaves correctly under such circumstances. Next, the content of the list is displayed with the use of the `iterate_list()` function invoked with an argument which is the address of the function that prints the value of a single element of the list. Please note, that the cursor is moved to the next line on the screen after the function exits. In the 4th line the same function is called with the address of the `double_element_value()` function passed as an argument. This time the `iterate_list()` function doubles the value of each of the elements of the list. Then, the content of the list is displayed on the screen using the `iterate_list()` function (please compare lines no. 2 and no. 5).

The `main()` Function

The Forth Part

Next, the `iterate_list_with_result()` function is tested. First it is called with the address of the `add_up()` function passed as an argument, and then with the address of the `count_element()` function. In the first case it sums up values of all the elements in the list. In the second one it counts the number of the elements. Eventually, the list is removed.

Summary

Using the recursion allows for shortening the definitions of functions that implement operations on the singly linked linear list. In the case of printing the content of the list on screen in the reversed order, the recursion greatly simplifies the implementation of such an operation. That allows for concluding that the recursion technique should be known to any decent programmer.
Examples of applications elements of functional programming for performing operations on the list are presented in the lecture. The aforementioned programming model gains recently on interest, because it makes it possible to avoid many issues with concurrent programming. Those problems are related to the imperative programming paradigm, which also means that they are present in the structural, procedural and object-oriented paradigms. The C language doesn't support directly the functional programming paradigm, thus the presented solutions are not "purely" functional, but still are worth studying.

Questions

?

The End

Thank You For Your Attention!