Fundamentals of Programming 2

Stack And Its Applications

Arkadiusz Chrobot

Department of Computer Science

March 2, 2020

Outline

1. Abstract Data Structures
2. Stack — Introduction
3. Stack — Implementation
4. Memory Leaks
5. Examples of Stack Applications
6. Summary

Abstract Data Structures

Thanks to the functions managing the heap, which were introduced in the previous lecture, it is possible to create dynamically allocated variables. However, there are many more possibilities offered by these functions. They allow the programmers to create the whole data structures called *dynamically allocated data structures*. The main advantage of such data structures is that their size doesn't have to be known before the program runs. They are created, as their name suggests, dynamically when the program is performed and their size is limited only by the available space in the heap. The *abstract data structures* are the majority of dynamically allocated data structures. Those structures usually are not a part of the programming language standard, but can be constructed with the help of elements provided by the language. To create such a structure, first its *abstract data type* has to be defined. The type describes not only the data that the structure can store but also the *operations* that can be applied to it. The *stack* is the first such a data structure that will be introduced in this course.

Stack — Introduction

A stack in computer science is an abstract data structure, that stores data according to the *Last In First Out* rule, or LIFO for short. It can be implemented in several different forms. One of them is an abstract data structures in which elements storing data are linked together with the use of pointers. The stack is also a special case of another data structures called *queues*, which in turn a special case of *lists*. All these structures will be successively introduced in the lectures. To better understand what is a stack, let's assume, that a list is a collection of connected elements with a beginning and an end. A stack is a kind of list in which operations of adding and removing elements can be applied to only one of its ends. Usually the stack is depicted as a vertical list which one end is called a *top*.

Stack — Implementation

The stack is an abstract data structures, so an abstract data type for the stack has to be created. The data type ought to define the type of information stored in the stack and also the operations that can be carried out on this structure. The task has to be completed with the use of elements provided by the C language. Its first part can be accomplished with a structure data type and the second with the functions. For the sake of simplicity let's assume, that the stack should store, in its elements, numbers of the `int` type. A stack storing strings of characters also will be presented in the lecture.

Stack — Implementation

Stack Base Type

```c
struct stack_node {
    int data;
    struct stack_node *next;
};
```

Stack — Implementation

The previous slide contains a definition of an example structure that describes the type of a single stack element which is also called the *stack base type*. It contains two fields. The first one stores data. In many cases there can be a larger number of such fields and they can store more complex data. In the example there is only one such a field of the `int` type and named `data`. Second field in the example base type is a pointer. There are stacks with elements containing more than one pointer field. However, in any dynamically allocated stack, at least one such a pointer field in each element should exist. Please notice the type of the pointer field. It is the same as the type of the structure that contains it. It means that the structure is *recursive* and that the pointer can point to another structure of the same type as the one in which it is contained. In other words, thanks to this pointer, the elements of stack can be linked together.

Stack — Implementation

Elements of the stack are linked together with the use of pointer fields. However, to perform an operation on the stack, the program has to know where is the top of the stack. Hence, a separate pointer, local or global, is needed for storing the address of the top of the stack. The pointer can have any name, but usually it is described as a `stack pointer`.
The only thing that is now missing in the definition of abstract type for the stack are the operations. The most basic of them are: adding a new element to the stack, which is called *push* and removing an element from the stack, which is named *pop*. Both those operations are performed on the top of the stack. An optional operation, which is sometimes defined, is retrieving the value of the stack top element and it is called *peek*. All three operations are defined in the lecture.

Stack — Implementation

The `push()` Function

The *push* operation is implemented in a form of **`push()`** function. The behaviour of the function should satisfy the following conditions (assertions):

1. Before the function is performed the stack pointer has to point the top element of the stack or be an empty pointer — in the latter case the stack is empty (or nonexistent).
2. The function as a result should return an address that either will be the same as the one stored in the stack pointer — in that case adding a new element to the stack has failed — or it will be an address of a new element on the top of the stack — in that case the operation has been successful.

Stack — Implementation

The push() Function

```
1  struct stack_node *push(struct stack_node *top, int number)
2  {
3      struct stack_node *new_node = (struct stack_node *)
4                  malloc(sizeof(struct stack_node));
5      if(new_node!=NULL) {
6          new_node->data = number;
7          new_node->next = top;
8          top = new_node;
9      }
10     return top;
11 }
```

Warning! The line numbers are not a part of the source code. They are introduced to make describing the function code easier.

Stack — Implementation

The `push()` Function

The `push()` function takes two arguments which are passed by its parameters. The first one is the stack pointer and the second is a number which is to be stored in a new element of the stack. First, the function allocates memory for the new element (lines no. 3 and 4). What happens next depends on the result of the allocation. If it fails then the `new_node` pointer value will be `NULL` and the function will return the unchanged value of the `top` pointer. Otherwise, the `new_node` pointer will store the address of the new element. The number passed by the `number` parameter will be stored in the `data` field of the new element (line no. 6). In the 7th line, the address currently stored in the `top` pointer in assigned to the `next` filed of the new element. Thus, the new element is linked to the rest of the stack and becomes a new top of the stack. Therefore its address is assigned to the `top` pointer in the 8th line. After that the function returns the address of a new top element of the stack and terminates.

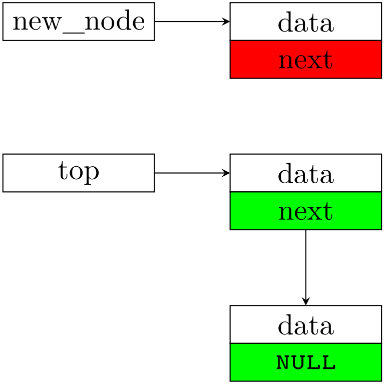
Stack — Implementation

The `push()` Function

The `push()` can be implemented in many was. Aside from the presented one, definitions exist in which the stack pointer is passed by a pointer to pointer parameter which is modified in the function body. Such a solution is discussed in more details in the `pop()` function description.
The next slides illustrate successful adding of a new element to the existing stack which has two elements. Please note, that the `next` field of the new element is initially marked in a red color. It means that it is an incorrect pointer (a wild pointer).

Stack — Implementation

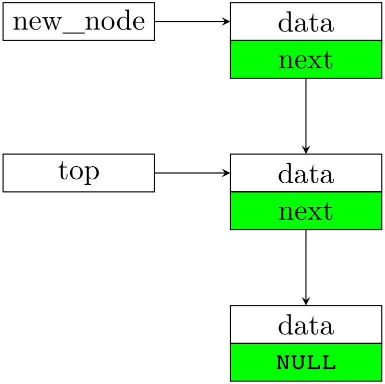
The `push()` Function



Before the 7th line of the `push()` function is performed.

Stack — Implementation

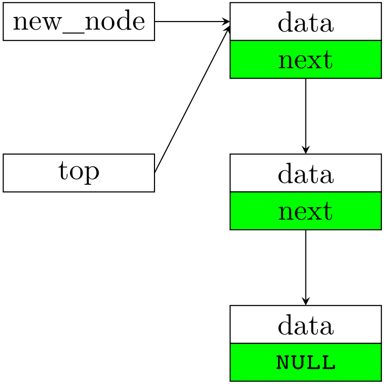
The `push()` Function



Before the 8th line of the `push()` function is performed.

Stack — Implementation

The `push()` Function



Before the 9th line of the `push()` function is performed.

Stack — Implementation

The `push()` Function

Please note the value of the `next` field in the *bottom* element of the stack. It is a `NULL` value. It means that the element which contains such a field is the last element of the stack. There are no other stack elements behind it. Let's find out if the `push()` function assures that the `next` field in the bottom element of the stack always gets the `NULL` value. It shows up, that it happens only when the function is given a stack pointer with the `NULL` value when it is called for the first time. In such a case the `NULL` value is assigned to the `next` field of the first and only element of the stack. However, if in such a case an incorrect pointer is passed to the function, then its value will be assigned to the `next` field. It is a dangerous situation from the program point of view, because it will be unable to locate the end of the stack. Thus, the programmers should always take care of passing an empty stack pointer to the `push()` function when it is creating a stack, i.e. when it is invoked for the first time in the program. This is especially important in case of local stack pointers.

Stack — Implementation

The `pop()` Function

The *pop* operation is implemented in the form of the `pop()` function. Similarly as in case of the `push()` function the behaviour of the `pop()` function should satisfy the following assertions:

1. Before the function is performed the stack pointer should point to the top element of an existing stack, or be an empty pointer.
2. After the function is performed, the stack pointer should point to the top element of the existing stack, which is one element shorter, or be an empty pointer.

Stack — Implementation

The `pop()` Function

```c
int pop(struct stack_node **top)
{
    int result = -1;
    if(*top) {
        result = (*top)->data;
        struct stack_node *tmp = (*top)->next;
        free(*top);
        *top = tmp;
    }
    return result;
}
```

Stack — Implementation

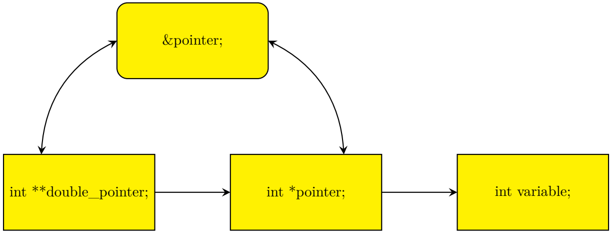
The `pop()` Function

The `pop()` function has only one parameter which is a *pointer to a pointer* or *double pointer*[1]. Using such a parameter is necessary because the function needs to modify the stack pointer and it is not possible to return its new value, because the `pop()` function also needs to return the value of `data` field of the removed stack element. In this case using the pointer to a pointer as a parameter is the best option. By this parameter the address of a stack pointer is passed to the function. The next slide illustrates how the pointer to pointer works.

[1] The latter name is sometimes confusing, because it can also mean a pointer of the `double` type.

Stack — Implementation

The `pop()` Function



Stack — Implementation

The `pop()` Function

The previous slide shows that the pointer to a pointer (`double_pointer`) can point to a pointer (`pointer`) which in turn points to a variable (`variable`). The variable can be statically or dynamically allocated. To access the value of the `variable` variable using the pointer to pointer the dereference operator have to be applied twice (like this: `**double_pointer`). If the operator is applied only once, then the value of the `pointer` pointer can be accessed.

Stack — Implementation

The `pop()` Function

The `pop()` function has a local variable (`result`) of the `int` type which initial value is set to `-1`. If the stack is empty then the function will return such a value and terminate. The state of the stack (empty or existing) is verified in the 4th line of the function. The `*top` condition is a shorter form of the `*top!=NULL` expression. If it's true the function assigns the value of the current top element of the stack to the `result` variable (line no. 5) and stores the address of a next element in the `tmp` pointer (line no. 6). The address is taken from the `next` field of the stack current top element. Then the top element is removed (line no. 7). Now, the stack pointer has an incorrect value — it doesn't point to the top of the stack. To fix it, in the 8th line the value of `tmp` pointer is assigned to the stack pointer. Next, the function returns the value of the `reslut` variable and terminates.

Stack — Implementation

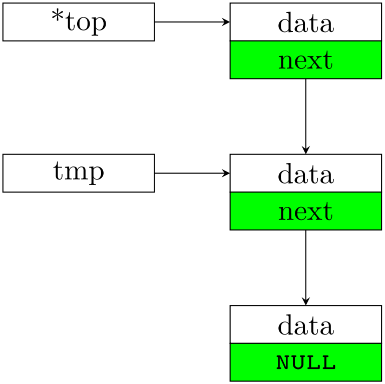
The pop() Function

Please observe, that the `pop()` function removes correctly also the last (the bottom) element of the stack. Its `next` pointer field has a value of `NULL` and such a value is assigned to the `tmp` when the 6th line of the function is applied to a stack with one and only element. After the 8th line is performed also the stack pointer gets such a value. This is an expected result, since the stack should become empty after its only element is removed.

The next slides illustrate the behaviour of the `pop()` function when it removes a top element from a stack that initially has three of them. Contrary to what the slide suggests the content of the removed element doesn't vanish after the `free()` function is called, nor the pointer that points to it becomes empty. Nonetheless the element should not be accessed any more. Also the pointer should not be used until a new address is stored in it. The reasons for such restrictions were explained in the previous lecture.

Stack — Implementation

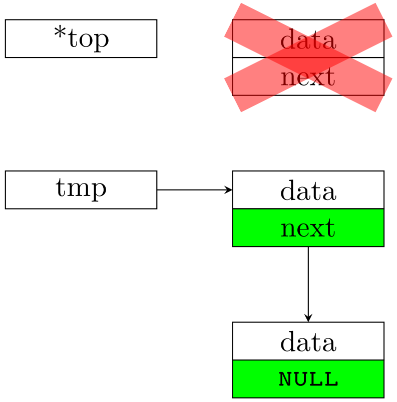
The `pop()` Function



After the 6th line of the `pop()` function is performed.

Stack — Implementation

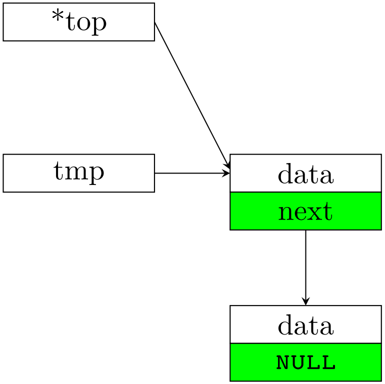
The `pop()` Function



After the 7th line of the `pop()` function is performed.

Stack — Implementation

The `pop()` Function



After the 8th line of the `pop()` function is performed.

Stack — Implementation

The peek() Function — Optional

```c
int peek(struct stack_node *top)
{
    if(top)
        return top->data;
    else {
        fprintf(stderr,"The stack is empty.\n");
        return -1;
    }
}
```

Stack — Implementation

The `peek()` Function — Optional

The *peek* operation is optional. It doesn't have to be implemented in every stack implementation. Nonetheless it is presented in this lecture in the form of a `peek()` function. Its definition in relatively simple. The stack pointer is passed to the function with the use of its parameter. If it is not empty (the condition `top` is shorter form of the expression `top!=NULL`), then the function returns the value stored in the top element of the stack (or the value of the element, for short). Otherwise, it prints a message on the screen informing the user that the stack is empty and returns the same value as the `pop()` function in the same case.

Stack — Implementation

Example Program

All elements necessary to use a stack are now implemented. The next slides present a simple program, that uses such a data structure. It generates subsequent natural numbers and stores them on the stack, then it gets them from the stack and displays on the screen. Aside from the header files the code of the program contains the functions that are defined in this lecture and the `main()` function. The latter function is the only one that needs to be described. Such a description is given in the slide that follows the slide with the `main()` function definition.

Stack — Implementation

Example Program

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

struct stack_node {
    int data;
    struct stack_node *next;
};
```

Stack — Implementation

Example Program

```
struct stack_node *push(struct stack_node *top, int number)
{
    struct stack_node *new_node = (struct stack_node *)
                        malloc(sizeof(struct stack_node));
    if(new_node!=NULL) {
        new_node->data = number;
        new_node->next = top;
        top = new_node;
    }
    return top;
}
```

Stack — Implementation

Example Program

```
int pop(struct stack_node **top)
{
    int result = -1;
    if(*top) {
        result = (*top)->data;
        struct stack_node *tmp = (*top)->next;
        free(*top);
        *top = tmp;
    }
    return result;
}
```

Stack — Implementation

Example Program

```
int peek(struct stack_node *top)
{
    if(top)
        return top->data;
    else {
        fprintf(stderr,"The stack is empty.\n");
        return -1;
    }
}
```

Stack — Implementation

Example Program

```c
int main(void)
{
    struct stack_node *top = NULL;
    srand(time(0));
    int i;
    for(i=1; i<6+rand()%5; i++)
        top=push(top,i);
    printf("The value of the stack top: %d\n",peek(top));
    while(top)
        printf("%d ",pop(&top));
    puts("");
    return 0;
}
```

Stack — Implementation

Example Program

The stack pointer is declared as a local variable of the `main()` function called `top`. Initially the stack is empty. In the `for` loop the `push()` function is called which adds elements to the stack. The values of the elements are defined by the loop counter (the `i` variable). The initial value of the counter is always `1`, but the final value is generated randomly and is ranging from `6` to `10`. Before the program runs it is hard to predict how many numbers will be stored in the stack. After the loop terminates the program displays the value of the stack top element on the screen. Next, the numbers are removed from the stack and printed on the screen in the `while` loop. The loop terminates when the stack is empty. Its condition is equivalent to the `top!=NULL` expression. The displayed numbers expose an important property of the stack: *elements of the stack are removed in reverse order to the one in which they were added.*

Memory Leaks

The implementations of dynamically allocated data structures are prone to serious errors. Incorrectly linked elements of a stack or similar structure are one of the examples. Let's assume that some overzealous programmer decides to zero out the `top` parameter at the beginning of the `push()` function. Such a mistake causes lack of connections between elements of the stack. Moreover, aside from the last element, non other is pointed by any pointer. Those elements cannot be deallocated. The areas of the heap that are allocated to the elements are lost until the program finishes. In the Computer Science jargon such a mistake is called a *memory leak*. In the worst case it can lead to exhaustion of the space in the heap. The first defence against memory leaks is to avoid them by carefully analysing implementations of all operations performed on the data structure. There exist also software tools like debuggers and dedicated libraries that make detecting of such errors easier. Unfortunately, they are not part of the C language standard, because their internal working depends on the used computer and operating system.

Stack — Applications

Conversion From Binary To Decimal

The stack is quite commonly used data structure. As a second example of its application a program is presented that converts a binary number to a decimal number. Using a stack implemented as a dynamically allocated data structure to this end is a suboptimal solution, but it is just for demonstrating the applications of the stack. In the next lecture a better solution of the problem will be presented. In the program the `getch()` function from the *curses* library is used to allow the program to read the binary number from the keyboard bit by bit. Each of the bits is stored in separated element of the stack. Hence, the least significant bit of the number is stored in the top element of the stack and the most significant one in the bottom element. Using the *curses* library requires to include the `curses.h` and `local.h` header files in the program.

Stack — Applications

Conversion From Binary To Decimal

```
#include<stdlib.h>
#include<curses.h>
#include<locale.h>

struct stack_node {
    int data;
    struct stack_node *next;
};
```

Stack — Applications

Conversion From Binary To Decimal

```
struct stack_node *push(struct stack_node *top, int number)
{
    struct stack_node *new_node = (struct stack_node *)
                        malloc(sizeof(struct stack_node));
    if(new_node!=NULL) {
        new_node->data = number;
        new_node->next = top;
        top = new_node;
    }
    return top;
}
```

Stack — Applications

Conversion From Binary To Decimal

```c
int pop(struct stack_node **top)
{
    int result = -1;
    if(*top) {
        result = (*top)->data;
        struct stack_node *tmp = (*top)->next;
        free(*top);
        *top = tmp;
    }
    return result;
}
```

Stack — Applications

Conversion From Binary To Decimal

```
void put_binary_on_stack(struct stack_node **top)
{
    int input = 0;
    do {
        input = getch();
        if(input=='0'||input=='1')
            *top=push(*top,input-'0');
    } while(input=='0'||input=='1');
}
```

Stack — Applications

Conversion From Binary To Decimal

The previous slides contains definitions of the `push()` and `pop()` functions which have been already discussed, so no new description of them is given. The last slide shows the definition of a function that reads successive characters entered by the user with the keyboard. If those characters are the `0` and `1` digits then it stores them in the stack. The activity is repeated until the user presses a key representing any other character. The function has only one parameter, which is a pointer to a pointer. By this parameter the stack pointer (initially empty) is passed to the function. The pointer is modified in the function. The characters are read from the keyboard in the `do…while` loop which terminates after a character is entered which is not a binary digit. Each bit is stored in a separate stack element. Please notice the `push()` function invocation. The value it returns is stored in the dereferenced `top` pointer which is also passed in the same form to the function as its first argument. The second argument is the `input - '0'` expression.

Stack — Applications

Conversion From Binary To Decimal

Because the value of the bit is stored in the `input` variable in a form of the ASCII code it has to be converted to a digit. It is accomplished by subtracting from the value the ASCII code of the `0` character.

Stack — Applications

Conversion From Binary To Decimal

```c
int convert_binary_to_decimal(struct stack_node **top)
{

    int result = 0, base = 1;
    while(*top) {
        int digit = pop(top);
        result += digit * base;
        base *= 2;
    }
    return result;
}
```

Stack — Applications

Conversion From Binary To Decimal

The actual conversion is performed by a function which definition is presented in the previous slide. The stack pointer is modified by the `pop()` function invoked in the described function. Hence, the latter function has a parameter which is a pointer to a pointer. The result of the conversion is stored in the local variable named `result`. Each bit removed from the stack, starting with the least significant, has to be multiplied by a corresponding base (the value of the `base` variable) which is a power of two ($2^0 = 1, 2^1 = 2, 2^2 = 4$, etc.). Hence, the value of the `base` variable is multiplied by `2` in each of the `while` loop iterations. The result of multiplying the removed bit by the corresponding base is added to the sum of such products calculated for the less significant bits. The loop terminates when the stack is empty. Next, the function returns the value of the `result` variable and terminates.

Stack — Applications

Conversion From Binary To Decimal

```
int main(void)
{
    if(setlocale(LC_ALL,"")==NULL) {
        fprintf(stderr,"Language settings initialization\
        exception!\n");
        return -1;
    }
    if(!initscr()) {
        fprintf(stderr,"The curses library initialization\
        exception!\n");
        return -1;
    }
```

Stack — Applications

Conversion From Binary To Decimal

The previous slide presents the beginning of the `main()` function which contains the code that initializes the language settings and the *curses* library.

Stack — Applications

Conversion From Binary To Decimal

```
printw("Please enter a binary number terminated with any\
character:\n");
(void)refresh();
struct stack_node *top = NULL;
put_binary_on_stack(&top);
```

Stack — Applications

Conversion From Binary To Decimal

In the part of `main()` function, presented in the previous slide, the message to the user informing her or him what he or she has to do is displayed on the screen and the `put_binary_on_stack()` function is invoked that creates a stack and stores in it the bits of binary number entered by the user.

Stack — Applications

Conversion From Binary To Decimal

```
    printw("The value of the binary number in decimal is: %d.\n",
                                convert_binary_to_decimal(&top));
    (void)refresh();
    getch();
    if(endwin()==ERR) {
        fprintf(stderr,"The endwin() function exception!\n");
        return -1;
    }
    return 0;
}
```

Stack — Applications

Conversion From Binary To Decimal

In the last part of the `main()` function, presented on the previous slide, the `convert_binary_to_decimal()` function is invoked, which converts the binary number to the decimal number. The obtained value is displayed on the screen and the program waits for the user to press any key. After the user does it, the *curses* library is finalized and the program finishes. Please note, that using the stack implemented as a dynamically allocated structure allows the program to convert a relatively big binary numbers. The limit is maximum number that can be stored in the `int` type variable, which stores the result of the conversion and the limit of the heap size.

Stack — Applications

Evaluating RPN Expressions

The stack is used for evaluating arithmetic expressions written in the *Reversed Polish Notation* (RPN) also called the *postfix* notation. The notation was proposed by an Australian computer scientist and philosopher Charles Hamblin and it is based on the *Polish Notation* (PN) also called the *prefix* notation proposed by Polish mathematician and philosopher Jan Łukasiewicz. Both notations do not require any parentheses to define the precedence of binary operators in any possible expression. In the PN the operators precede the arguments and in the RPN they follow the arguments. The next slide presents several expressions written in the traditional (infix) notation and in the RPN.

Stack — Applications

Evaluating RPN Expressions

$2 + 2 \Rightarrow 2\ 2\ +$

$(5 - 2) * (4 + 1) \Rightarrow 5\ 2\ -\ 4\ 1\ +\ *$

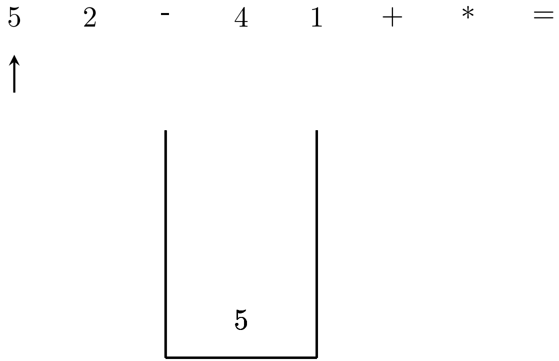
$(3 + 2) * 7 \Rightarrow 3\ 2\ +\ 7\ *$

$3 + 2 * 7 \Rightarrow 2\ 7\ *\ 3\ +$

Stack — Applications

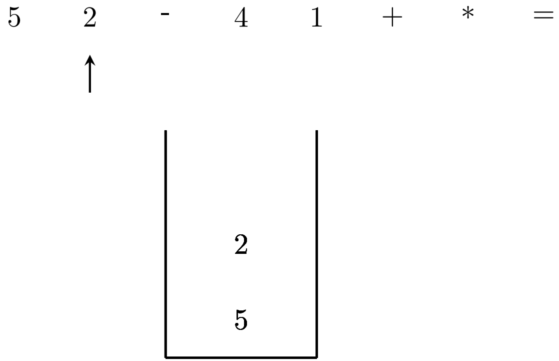
Evaluating RPN Expressions

The animation shows how a value of an RPN expression can be evaluated with the use of the stack.



Stack — Applications
Evaluating RPN Expressions

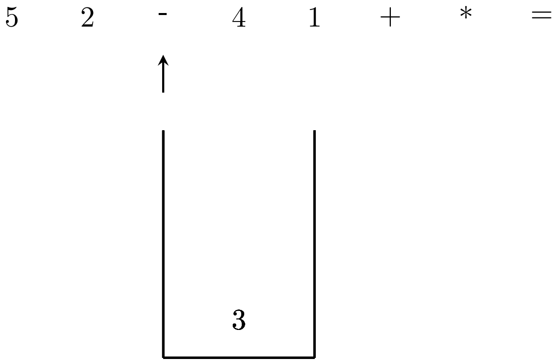
The animation shows how a value of an RPN expression can be evaluated with the use of the stack.



Stack — Applications

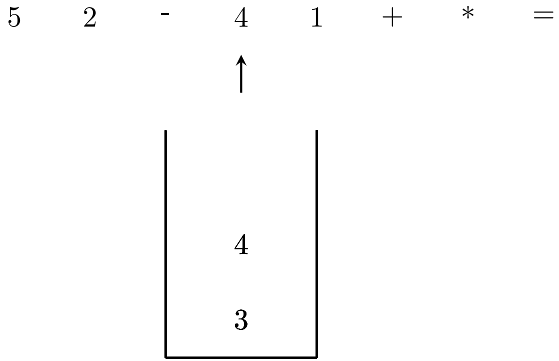
Evaluating RPN Expressions

The animation shows how a value of an RPN expression can be evaluated with the use of the stack.



Stack — Applications
Evaluating RPN Expressions

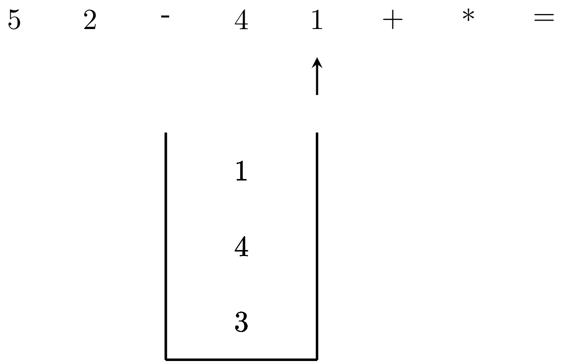
The animation shows how a value of an RPN expression can be evaluated with the use of the stack.



Stack — Applications

Evaluating RPN Expressions

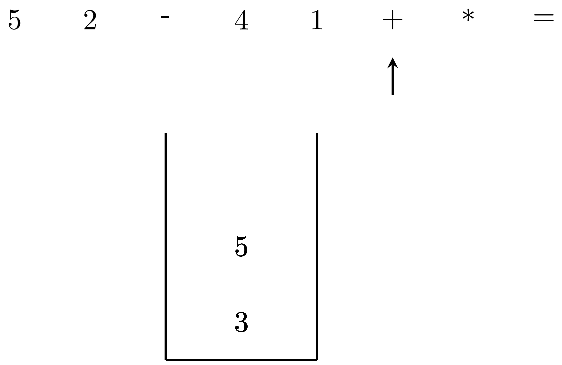
The animation shows how a value of an RPN expression can be evaluated with the use of the stack.



Stack — Applications

Evaluating RPN Expressions

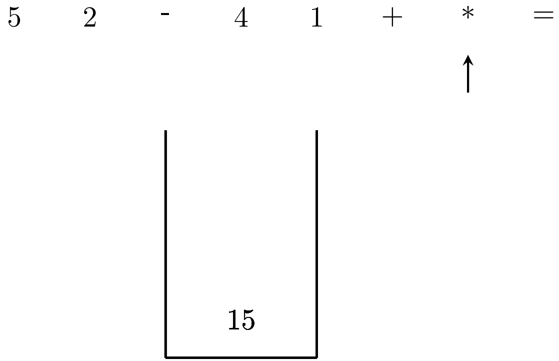
The animation shows how a value of an RPN expression can be evaluated with the use of the stack.



Stack — Applications

Evaluating RPN Expressions

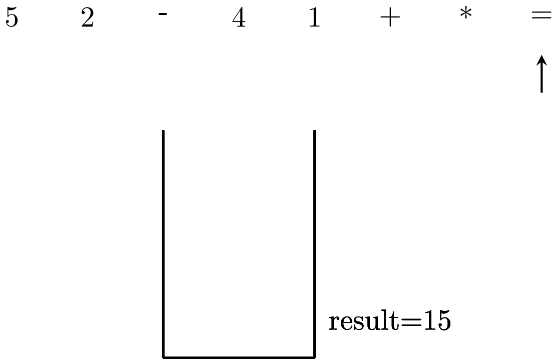
The animation shows how a value of an RPN expression can be evaluated with the use of the stack.



Stack — Applications

Evaluating RPN Expressions

The animation shows how a value of an RPN expression can be evaluated with the use of the stack.



Stack — Applications

Evaluating RPN Expressions

As it can be observed in the animation from the previous slide, the RPN expressions are read from the left to the right side. If a symbol is met that is a number, then it is added to the stack, but if it is an operator, then its arguments are removed from the stack (if the RPN expression is correct, then a proper number of arguments is already stored on the stack), the operation is carried out and its result is stored back in the stack. The example program implements an evaluation of very crude RPN expressions, in particular:

1. the RPN expressions consist only of single-digit natural numbers and three types of operators: adding, multiplying and subtracting,
2. the RPN expressions do not contain any whitespaces,
3. the `=` symbol terminates every RPN expression and informs the program to start evaluating it,
4. the program doesn't check the correctness of the RPN expression — it assumes that the expression is correct.

Stack — Applications

Evaluating RPN Expressions

The beginning of the program is the same as in the program that converts binary number to decimal — the described program also uses the *curses* library and the `push()` and `pop()` functions.

Stack — Applications

Evaluating RPN Expressions

```
#include<stdlib.h>
#include<curses.h>
#include<locale.h>

struct stack_node {
    int data;
    struct stack_node *next;
};
```

Stack — Applications

Evaluating RPN Expressions

```
struct stack_node *push(struct stack_node *top, int number)
{
    struct stack_node *new_node = (struct stack_node *)
                        malloc(sizeof(struct stack_node));
    if(new_node!=NULL) {
        new_node->data = number;
        new_node->next = top;
        top = new_node;
    }
    return top;
}
```

Stack — Applications

Evaluating RPN Expressions

```c
int pop(struct stack_node **top)
{
    int result = -1;
    if(*top) {
        result = (*top)->data;
        struct stack_node *tmp = (*top)->next;
        free(*top);
        *top = tmp;
    }
    return result;
}
```

Stack — Applications

Evaluating RPN Expressions

```
int calculate_rpn_expression(void)
{
    int input = 0;
    struct stack_node *top = NULL;
    do {
        input = getch();
        int first_argument=0, second_argument=0, result=0;
        switch(input) {
        case '+':
            result = pop(&top) + pop(&top);
            top = push(top,result);
            break;
        case '-':
            first_argument = pop(&top);
            second_argument = pop(&top);
            top = push(top,second_argument - first_argument);
            break;
```

Stack — Applications

Evaluating RPN Expressions

```
            case '*':
                result = pop(&top)*pop(&top);
                top = push(top,result);
                break;
            default:
                if(input>='0'&&input<='9')
                    top=push(top,input-'0');
            }
        } while(input!='=');

        return pop(&top);
    }
```

Stack — Applications

Evaluating RPN Expressions

The parameterless `calculate_rpn_expression()` function presented in the two previous slides reads in the `do…while` loop subsequent characters, entered by user with the use of the keyboard, which belong to the RPN expression and recognizes them. The recognition of expressions in Computer Science is called *parsing*. If the character from the keyboard is a digit, then the program stores its numeric value in the stack. However, it the characters is an operator then the function removes two of its arguments from the stack[2], carries out the recognized operation and stores the result back in the stack. The loop terminates when the “equals” symbol is recognized. After that the function returns the only value that is stored in the stack and also terminates. The stack should now be empty.

[2] **Be careful with the order of the arguments for the subtraction operator!**

Stack — Applications

Evaluating RPN Expressions

```c
int main(void)
{
    if(setlocale(LC_ALL,"")==NULL) {
        fprintf(stderr,"Language settings initialization exception!\n");
        return -1;
    }
    if(!initscr()) {
        fprintf(stderr,"The curses library initialization exception!\n");
        return -1;
    }
    printw("Please enter an RPN expression:\n");
    (void)refresh();
    printw("\nThe result is: %d.\n",calculate_rpn_expression());
    (void)refresh();
    getch();
    if(endwin()==ERR) {
        fprintf(stderr,"The endwin() function exception!\n");
        return -1;
    }
    return 0;
}
```

Stack — Applications

Evaluating RPN Expressions

In the program's `main()` function, aside from the functions that initialize and finalize the *curses* library and change the language settings, also the `calculate_rpn_expression()` function is invoked. It returns the value of the RPN expression as its result. The length of the expression is limited only by the size of the available free memory in the heap. It is due to the use of a stack in a form of a dynamically allocated data structure. The value of the expression has to fit in the range of the `int` data type.

Stack — Applications

Stack of Strings

The last example shows how to use the stack for storing strings of characters. It is assumed that the number of the characters in a single string is limited to 100. The program doesn't use the *curses* library. The strings are passed to the program as its arguments. In the program aside from the `stdio.h` and `stdlib.h` header files also the header file associated with string processing is included.

Stack — Applications

Stack of Strings

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

typedef struct stack_node {
    char arg[100];
    struct stack_node *next;
} node;
```

Stack — Applications

Stack of Strings

In the program the declaration of the first field is modified so it can store string of at most 100 characters. Also its name is changed. The `typedef` keyword is applied in order to avoid repeating the whole specification of the pointer type in the definitions of functions. It's a convenient albeit less legible solution.

Stack — Applications

Stack of Strings

```
node *push(node *top, char *str)
{
    node *new_node = NULL;
    new_node = (node*)malloc(sizeof(node));
    if(new_node==NULL) {
        fprintf(stderr,"A memory allocation exception!\n");
        return top;
    }
    strncpy(new_node->arg,str,100);
    new_node->next=top;
    return new_node;
}
```

Stack — Applications

Stack of Strings

The `push()` function is defined in a similar fashion as in the previously presented programs, but as a second argument it takes a pointer to a string. The string is copied with the use of `strncpy()` function to the new element of the stack. Please note, that in case of the memory allocation failure, the function first displays a message and only then it returns the address of the original stack top element. Also returning of the new element address is implemented in a little more compact manner — after the new element is linked to the rest of the stack its address is immediately returned by the function. The address should be assigned to the stack pointer in the place in code where the function is invoked.

Stack — Applications

Stack of Strings

```
node *pop(node **top)
{
    node *next = NULL, *old_top = NULL;
    if(*top!=NULL) {
        next=(*top)->next;
        (*top)->next = NULL;
        old_top = *top;
        *top = next;
    }
    return old_top;
}
```

Stack — Applications

Stack of Strings

The `pop()` function presented in the previous slide doesn't free the memory allocated to the stack top element, but it only disconnects the element from the rest of the stack, assigns the `NULL` value to its `next` pointer field and returns its address. The stack pointer is passed to the function by the pointer to a pointer parameter. Inside the function the stack pointer is modified, so after the current stack top element is unlinked from the rest of the stack, the address of the element that followed it in the stack is stored in that pointer.

Stack — Applications

Stack of Strings

```c
int main(int argc, char **argv)
{
    node *top = NULL;
    int i;
    for(i=0; i<argc; i++)
        top = push(top,argv[i]);
    while(top) {
        node *tmp = pop(&top);
        printf("%s\n",tmp->arg);
        free(tmp);
        tmp=NULL;
    }
    return 0;
}
```

Stack — Applications

Stack of Strings

In the `main()` function the program arguments are stored in the stack inside the `for` loop. Because the argument stored in the first element of the `argv` array always exists — it is the full path to the program executable file, then the program always displays at least one message on the screen. Elements of the stack are removed in the `while` loop. Please observe, that they are deallocated one by one, outside the `pop()` function.

Summary

The stack has many more applications than it is presented in the lecture. The compilers uses it to evaluate the values of arithmetical expressions. The algorithm for converting those expressions from the infix to the postfix notation also uses a stack, but this time the operators and parentheses, instead of the numbers are stored there. The author of this algorithm is Edsgar Dijkstra and it is called a *shunting-yard algorithm*. It won't be however discussed in details in the lecture. The operating systems use the stack for managing processes and resources. User applications, such as text editors use the stack for implementing the "undo" operation.

Summary

The stack can be implemented with the use of a “regular”, i.e. statically allocated array. However, its size is that case is limited by the size of the array. As a stack pointer the index of the array can be used. Adding an element to the stack involves incrementing the value of the index by one and storing the data in the element of the array designated by the index. Removing an element from the stack consists of reading the data from the array element designated by the index and decreasing the value of the index by one.

Questions

?

The End

Thank You For Your Attention!